X.systems.press

X.systems.press ist eine praxisorientierte
Reihe zur Entwicklung und Administration von
Betriebssystemen, Netzwerken und Datenbanken.

Christoph Czernohous

Pervasive Linux

Basistechnologien, Softwareentwicklung, Werkzeuge

Springer

Christoph Czernohous
Silcherplatz 5
71106 Magstadt
Deutschland
cc@de.ibm.com

ISSN 1611-8618
ISBN 978-3-540-20940-9 e-ISBN 978-3-540-68426-8
DOI 10.1007/978-3-540-68426-8
Springer Heidelberg Dordrecht London New York

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Einbandentwurf: KünkelLopka GmbH, Heidelberg

Gedruckt auf säurefreiem Papier

Springer ist Teil der Fachverlagsgruppe Springer Science+Business Media (www.springer.com)

Für Bettina

Vorwort

Pervasive heißt auf Deutsch so viel wie durchdringend, allgegenwärtig, sich überall ausbreitend. Und genau das geschieht zur Zeit mit der Rechenleistung von Computern aller Art. Es gibt kaum noch technische Alltagsgegenstände, die nicht irgendeine Form von Mikroprozessor beinhalten, die diese Geräte steuern und zusätzliche Funktionalität bereitstellen, von der Waschmaschine über das Auto zum Mobiltelefon. Wir werden jeden Tag fast überall von Rechenleistung umgeben, sie ist allgegenwärtig und durchdringt unser Leben, auch wenn wir es manchmal gar nicht merken.

Zur gleichen Zeit gibt es einen weiteren Trend mit starkem Wachstumspotenzial: Open Source. Die Verbreitung von Software, deren Quellen für jedermann einsehbar sind, hat in den letzten Jahren stark zugenommen, allem voran das offene Betriebssystem Linux. Ausgehend von einem Betriebssystem für Personal Computer, hat sich Linux in der Zwischenzeit ein festes Standbein im Server-Umfeld geschaffen und wird auch in eingebetteten Systemen, der Grundlage für Pervasive Computing, immer beliebter.

Der Kombination dieser beiden Wachstumsperspektiven, Pervasive Computing und Linux, gilt das vorliegende Buch. Es richtet sich an Produkt- und Softwareentwickler, die Linux in eingebetteten Systemen einsetzen möchten, setzt dabei aber Grundkenntnisse über die Benutzung und Administration von Linux, sowie Programmierkenntnisse voraus. Auch die Vor- und Nachteile von Open Source im allgemeinen und Linux im besonderen werden nicht erörtert. Dieses Buch nähert sich dem Thema Pervasive Linux von der Anwendungs- bzw. Softwareentwicklungsseite. Es geht hauptsächlich darum, den Einstieg in die Entwicklung für Pervasive Linux zu erleichtern. Angesichts der Vielfalt an verfügbarer Hard- und Software in diesem Bereich sei für das vertiefende Studium einzelner Themen auf die jeweils spezialisierte Literatur verwiesen. Nicht Gegenstand des Buches ist die Portierung von Linux selbst auf zusätzliche Hardware-Plattformen. Aus den selben Gründen werden auch nicht sämtliche Parameter diverser Kommandozeilenbefehle und Konfigurationsdateien aufgelistet.

Oft ist die grösste Hürde, überhaupt den Einstieg in Pervasive Linux zu finden. Der Aufbau einer funktionsfähigen Entwicklungsumgebung kann sehr frustrierend sein. Oftmals sind Informationen auf den relevanten Webseiten nicht sofort auffindbar. Dieses Buch soll den Einstieg in die Entwicklung von Linux-Anwendungen für

das Pervasive Computing erleichtern, indem es jeweils mit Beispielen die Grundlagen für die benötigten Schritte erläutert. Dabei werden auch Werkzeuge vorgestellt, die nicht spezifisch für Pervasive Linux sind, aber bei der Benutzung und Entwicklung häufig verwendet werden.

Es wird erklärt, welche Besonderheiten beim Einsatz von Linux in eingebetteten Systemen zu beachten sind, welche Code- und Informationsquellen es gibt, welche Aufgaben ein eingebettetes Linuxsystem zu erfüllen hat und aus welchen Bestandteilen es besteht. Die Softwareentwicklung wird mit Hilfe gängiger Werkzeuge beispielhaft dargestellt.

Die Beispiele orientieren sich an Linux-Systemen, die auf tragbaren digitalen persönlichen Organisationshilfen (engl.: Personal Digital Assistants – PDA, bzw. Handhelds), mobilen Internetgeräten (engl.: Mobile Internet Device – MID), Tablett-Computern und Mobiltelefonen bzw. so genannten Smartphones einsetzbar sind, da diese Art Hardware-Grundlage als Massenware relativ unkompliziert und günstig für jedermann erhältlich ist und es für diese Systeme oft eine breite Linux-Unterstützung gibt. Die Konzepte lassen sich aber auch auf andere eingebettete Systeme übertragen.

Das Motto des Buches lässt ich also wie folgt formulieren: *Linux macht Spass. Pervasive Linux macht den Spass allgegenwärtig*.

Danksagung

Mein Dank gilt meiner Frau Bettina und meinem Redakteur Hermann Engesser vom Springer Verlag für die schier endlose Geduld und Unterstützung bei diesem Buchprojekt.

Ausserdem möchte ich mich bei Silke und Clemens van Dinther, Jan Burchhardt und Jens Kretzschmar fürs Korrekturlesen des Manuskripts und die konstruktiven Anregungen bedanken.

Magstadt, Deutschland Christoph Czernohous
Juni 2011

Inhaltsverzeichnis

Teil I
Systembestandteile

Kapitel 1
Pervasive Computing

Bei *Pervasive Computing* geht es hauptsächlich um kleine, verteilte, auf unterschiedlichster Hardware basierende Computer, die auf diverse Weise mit anderen Geräten oder dem Menschen kommunizieren. Oft ist auch von eingebetteten Systemen (engl.: Embedded Systems) oder so genannter Rechenallgegenwart (engl.: Ubiquitous Computing) die Rede.

Systeme des Pervasive Computing besitzen besondere Eigenschaften, die es bei deren Einsatz und insbesondere bei der Software-Entwicklung für diese Systeme zu berücksichtigen gilt. Besonders für Anwender und Entwickler, die aus dem Bereich des Desktop Computing kommen, gibt es einige zusätzliche Aspekte zu beachten.

1.1 Ressourcenbeschränkung

Wie in allen Bereichen der Informationstechnik steigt auch die Leistungsfähigkeit von eingebetteten Geräten immer weiter, während gleichzeitig die physikalischen Bauteile kleiner und integrierter werden. Nichtsdestotrotz sind die in diesem Umfeld zur Verfügung stehenden Betriebsmittel und Ressourcen im Vergleich zu gängigen Arbeitsplatzrechnern deutlich beschränkt. Das betrifft in erster Linie die Leistungsfähigkeit der verwendeten Prozessoren und den verfügbaren Speicher (sowohl Arbeits- als auch Ablagespeicher), aber auch Kommunikations- und Netzwerkverbindungen sowie die Energieversorgung.

All diese Aspekte spielen im Pervasive Computing eine zentrale Rolle und müssen bei der Entwicklung solcher Systeme bedacht werden.

1.1.1 Speicherverwaltung

Für die Verwaltung des Zugriffs auf Arbeitsspeicher besitzen Prozessoren üblicherweise eine Speicherverwaltungseinheit (engl.: Memory Management Unit – MMU). Die Aufgabe einer MMU ist es, laufenden Prozessen Arbeitsspeicher zur Verfügung zu stellen, der für einen konkreten Prozess exklusiv reserviert ist, so dass andere Prozesse den Inhalt des Speicherbereiches nicht auslesen oder verändern können.

C. Czernohous, *Pervasive Linux*, X.systems.press,
DOI 10.1007/978-3-540-68426-8_1, © Springer-Verlag Berlin Heidelberg 2012

Für das in einem Prozess aufgeführte Programm erscheint der so bereitgestellte Speicherbereich als ein kontinuierlicher Block. Tatsächlich kann sich der Speicherbereich aber über unterschiedlichste unzusammenhängende Speicheradressen und auch ausgelagerten Speicher, z. B. auf einer Festplatte, erstrecken.

Aus Platz- und Energiegesichtspunkten wird bei manchen Prozessoren auf eine MMU verzichtet. Wenn auf einem solchen System ein Betriebssystem betrieben werden soll, das nebenläufigen Prozessen exklusiven Speicher zur Verfügung stellt, wie das bei Linux der Fall ist, muss die Speicherverwaltungseinheit im Betriebssystem implementiert sein. Abschn. 3.2.3 beschäftigt sich mit der Unterstützung von Linux für solche Systeme.

1.1.2 Grafische Benutzerschnittstellen

Während man im Umfeld von Desktop-Computern viele Jahre lang von immer grösseren Bildschirmen mit entsprechender Auflösung und einheitlichem Seitenverhältnis ausging, erfordert die Entwicklung von Software für Geräte des Pervasive Computing ein Umdenken und die Berücksichtigung einiger Besonderheiten, wie z. B. variable Bildschirmgrössen und unterschiedliche Interaktionsmöglichkeiten mit dem Benutzer.

Für Computerprogramme, die sich in grafische Benutzeroberflächen (engl.: Graphical User Interface – GUI) integrieren, sollte es selbstverständlich sein, auf unterschiedliche und sich potenziell ändernde Umgebungen flexibel reagieren zu können. Diese Regel verschärft sich für das Pervasive Computing insofern, als der verfügbare sichtbare Bereich meistens sehr beschränkt ist, und durch weitere eingeblendete Bereiche, wie z. B. eine virtuelle Tastatur, zusätzlich eingeschränkt wird.

Ein weiterer wichtiger Punkt, den es zu berücksichtigen gilt, sind die unterschiedlichen Wortlängen, die Texte in verschiedenen Sprachen aufweisen. Daher sollte bei der Gestaltung von Oberflächen nicht von den Dimensionen der primär unterstützten Sprache ausgegangen werden. Das gilt natürlich nicht nur für das Pervasive Computing, aufgrund der eingeschränkten Platzverhältnisse verschärft sich hier aber die Problematik. In Abschn. 6.8 wird auf die technischen Möglichkeiten eingegangen, Anwendungen für unterschiedliche Sprach- und Kulturkreise zu entwickeln.

1.1.2.1 Variable Bildschirmgrössen

Die Anzeigefläche, die für Programme zur Verfügung steht, variiert zwischen unterschiedlichen Geräten stark. Nicht nur das Platzangebot selbst, also die Grösse des Bildschirms, sondern auch das Seitenverhältnis in Kombination mit der Ausrichtung muss bei der Entwicklung beachtet werden. Moderne Geräte reagieren mit einem Bewegungssensor darauf, wie das Gerät momentan eingesetzt wird, also ob es waagerecht oder senkrecht gehalten wird und welche Seite oben ist. Entsprechend wird das Seitenverhältnis angepasst und Anwendungen müssen darauf reagieren. Anwendungen sollten möglichst wenig Annahmen über den verfügbaren Raum machen und flexibel auf Änderungen reagieren.

Da sich die Grösse von Fenstern und Dialogen durch Modifikation des Benutzers jederzeit ändern kann, und auch Anzeigeflächen unterschiedliche gross sein können, bedarf dies bei der Entwicklung besonderer Aufmerksamkeit. Insofern sollten Steuerelemente wie Auswahlknöpfe und Eingabefelder möglichst dynamisch innerhalb eines Fensters angeordnet werden.

1.1.2.2 Interaktion und Eingabemöglichkeiten

Mobile Geräte des Pervasive Computing sind häufig mit einem berührungsempfindlichen Bildschirm (engl.: Touch Screen) ausgestattet. In vielen Fällen wird dann auf eine separate Tastatur verzichtet und die Eingabe komplett über den Bildschirm gesteuert, oft mit Hilfe eines speziellen Stiftes, mit welchem sehr genaue Eingabepunkte möglich sind. Allerdings werden für die Eingabe im mobilen Umfeld häufig auch die Finger verwendet. Bei Anwendungen, die mit den Fingern bedient werden sollen, muss darauf geachtet werden, dass die angezeigten Steuerelemente ausreichend grossflächig sind, da die Positionierung mit den Fingern deutlich ungenauer als mit einem Spezialstift ist.

1.2 Grafik

Für die grafische Darstellung von Informationen gibt es die Möglichkeit der punktgenauen Wiedergabe (Bitmap-Grafik) oder der beschreibenden Visualisierung (Vektorgrafik).

1.2.1 Vektorgrafik

Bei Abbildungen auf Bildschirmen unterschiedlicher Grösse und Auflösung kann durch den Einsatz von Vektorgrafiken eine einheitliche Darstellung erreicht werden. Dabei wird die darzustellende Grafik abhängig von den Anzeigemöglichkeiten berechnet. Das hat zwar eine erhöhte Rechenzeit zur Folge, sorgt aber für gleichmässigere Visualisierung auf unterschiedlichen Geräten, was gerade bei den unterschiedlichen Anzeigeformaten mobiler Geräte wichtig ist. Ein weiterer Vorteil von Vektorgrafiken ist der in der Regel geringe Speicherbedarf der Bildinformationen im Vergleich zu pixelbasierten Lösungen.

1.2.2 Bitmap-Grafik

Bei Bitmap-Grafiken wird die Anzeigeinformation Punkt für Punkt in der Bilddatei abgelegt. Bitmap-Grafiken könnten daher sehr schnell geladen und angezeigt werden, besitzen aber immer die gleichen physikalischen Ausmasse und variieren daher in Grösse und Lesbarkeit in Abhängigkeit des eingesetzten Anzeigegerätes.

Wie in Abschn. 1.1.2.1 bereits beschrieben, ist das eine besondere Herausforderung für Systeme mit unterschiedlichen Darstellungsgrössen, der sich Projekte wie z. B. Enlightenment (Abschn. 3.6.6) angenommen haben. Da die Bildinformation für jeden Bildpunkt eindeutig abgelegt wird, steigt der Speicherbedarf proportional zur Bildgrösse.

1.3 Mobilität

Mobiltelefone, digitale Organisationshilfen für Adress- und Terminverwaltung (so genannte *Personal Digital Assistants – PDA*), MP3-Spieler etc. sind dem Pervasive Computing zuzurechnen. Derartige Geräte bleiben aufgrund ihres für sie vorgesehenen Einsatzzwecks nicht fest an einem Ort, sondern werden in sich laufend verändernden Umgebungen eingesetzt. Die Software auf solchen Geräten muss deutlich flexibler und fehlertoleranter z. B. mit einer unterbrochenen Netzwerkverbindung oder Stromversorgung oder sich ändernden Uhrzeiten und Zeitzonen umgehen können.

Dadurch erhöhen sich die Anforderungen bzgl. der Datensicherheit dieser Geräte.

Ein weiterer Aspekt der Mobilität dieser Geräte ist die Möglichkeit, den aktuellen Standort auf der Erde zu bestimmen. Auf diverse Weise ist es heute möglich, die aktuelle Position sinnvoll in einer Anwendung zu verarbeiten.

1.3.1 Kontextverarbeitung

Die zunehmende Vernetzung und die modernen Möglichkeiten im Bereich des Pervasive Computing erfordern eine stärkere Berücksichtigung des Anwendungskontextes, als es bei herkömmlichen Anwendungen der Fall ist.

Bei einem MP3-Spieler ist es gleichgültig, in welcher Zeitzone er betrieben wird. Bei einem PDA ist diese Information schon wichtiger, damit z. B. Termine der momentanen Zeitzone entsprechend angezeigt werden. Mit aktuellen Mitteln wie z. B. der so genannten Geolocation, also der Standortbestimmung, nimmt die Bedeutung des Anwendungskontextes noch zu. Beispielsweise kann über entsprechende Dienste der aktuelle Standort an bekannte Personen verteilt werden, und ein mobiles Gerät so zusätzlichen Nutzen bieten. Weitere Beispiele sind das Erkennen von Systemen in der Umgebung, mit denen potenziell kommuniziert werden kann. Z. B. kann sich ein mobiles Gerät automatisch mit so genannten WLAN-Hotspots (Abschn. 4.4) eines bestimmten Betreibers verbinden, um eine Internetverbindung aufzubauen oder über Bluetooth (Abschn. 4.6) zusätzliche Funktionalität bereitstellen.

Je mehr Informationen ein Gerät über die aktuelle Umgebung liefern kann, desto anspruchsvoller wird die Auswertung dieser Daten, aber desto höher ist auch der potenzielle zusätzliche Nutzen für den Anwender.

1.3.2 Datensicherheit

Die im Vergleich zur herkömmlichen Datenverarbeitung deutlich erhöhte Mobilität im Pervasive Computing erfordert es, ein besonderes Augenmerk auf Datensicherheit und der Vermeidung von Datenverlust zu legen. Datensicherheit hat diverse Aspekte. Einserseits geht es um Datenverlust im Allgemeinen, andererseits um die Sicherheit der gespeicherten Daten vor unbefugtem Zugriff.

1.3.2.1 Datenabgleich und Synchronisierung

Da bei mobilen Geräten der unterbrechungsfreie Betrieb oft nicht garantiert werden kann, z. B. aufgrund mangelnder Energieversorgung, sollten geänderte Nutzdaten so schnell wie möglich vom Hauptspeicher auf ein persistentes Speichermedium gesichert werden, um Datenverlust zu vermeiden.

Mobilgeräte sind aufgrund ihrer Grösse und ihres Einsatzzwecks in erhöhtem Masse verlustgefährdet, sei es durch Unachtsamkeit, Diebstal oder Zerstörung. Meistens können Hard- und Software in solchen Fällen relativ einfach ersetzt werden, das eigentlich Wertvolle, nämlich die lokal gespeicherten Daten, sind dabei deutlich aufwändiger, wenn überhaupt, wieder herzustellen. Es ist daher wichtig, eine tragfähige Strategie zur Erstellung von Sicherheitskopien (engl.: Backup) z. B. mittels Synchronisation zu etablieren.

Insbesondere mobile Geräte, selbst wenn sie wie PDAs oder Smartphones nicht auf den Abgleich ihrer Daten mit anderen, zentralen Systemen angewiesen sind, sollten ihren Datenbestand regelmässig mit einem Sicherungssystem synchronisieren. Wenn, wie im beim Pervasive Computing häufig der Fall, ein ständiger Datenaustausch nicht möglich ist, weil keine Netzwerkverbindung besteht, müssen bestimmte Daten zwischengespeichert und, sobald eine Netzverbindung vorhanden ist, mit der Bestandsdatenbank abgeglichen werden.

All das erfordert die Möglichkeit, Daten lokal persistent zu speichern, und bei der Verfügbarkeit einer Netzverbindung, diese mit dem zentralen Datenbestand abzugleichen. Gerade für moderne Web-Anwendungen (Abschn. 6.7.2) ist das eine besondere Herausforderung.

1.3.2.2 Zugriffskontrolle und Verschlüsselung

Ein weiterer Aspekt ist der unbefugte Zugriff auf Daten und Funktionalität.

Das Ausspähen von Daten, die über eine Netzwerkverbindung übertragen werden, oder der Zugriff auf Daten bei Verlust des Gerätes konfrontieren Anwender und Entwickler von Software für Pervasive Computing mit erhöhten Sicherheitsanforderungen.

Passwortschutz und geeignete Verschlüsselungsmechanismen können die Datensicherheit erhöhen.

Kapitel 2
Hardware

Die Hardware, die zusammen mit Linux zum Einsatz kommt – bzw. kommen kann – reicht von Grossrechnern, bis zu teilweise winzigen Geräten, dem Thema dieses Buches. Je kleiner die Systeme und je günstiger die Produktionskosten sind, desto grösser ist auch die Anzahl an Varianten, die von Linux unterstützt werden. Von Seiten der Hardware sind dem Einsatz von Linux fast keine Grenzen gesetzt.

Daher ist es wichtig, mit der breiten Palette der Hardware-Bestandteile vertraut zu sein, um deren Auswahl, Einsatz und Eigenschaften beurteilen zu können.

2.1 Prozessor

Im Umfeld des Pervasive Computing kommen häufig Prozessoren auf Grundlage der so genannten *Reduced Instruction Set Computer* (*RISC*) Architektur zum Einsatz. RISC Prozessoren haben im Gegensatz zu Prozessoren auf Basis eines komplexen Befehlsumfangs (engl.: *Complex Instruction Set Computer* (*CISC*)) einen stark reduzierten Befehlssatz, der wiederum sehr effizient implementiert ist. Dadurch wird die Prozessorstruktur einfach gehalten. Ein RISC Prozessor benötigt in der Regel deutlich weniger Energie als ein CISC Prozessor, was besonders bei mobilen Einheiten wie PDAs und Mobiltelefonen wichtig ist, da diese nicht über eine ständige Stromversorgung verfügen und in den meisten Fällen auf Batterie- oder Akkustrom angewiesen sind, mit dem es möglichst effizient umzugehen gilt.

Allerdings drängen inzwischen auch Prozessoren mit mehreren Rechenkernen in den Bereich des Pervasive Computing vor. Damit wird es möglich, Programme echt parallel auszuführen, was den Entwickler insbesondere dann vor neue Herausforderungen stellt, wenn diese Parallelität Einfluss auf das Programmverhalten zum Ausführungszeitpunkt hat.

Für RISC Prozessoren übersetzte Programme sind in Binärform bis auf die einfachsten Fälle immer größer als der für CISC Prozessoren entsprechend übersetzte Code, da die fehlenden Prozessorbefehle durch Software abgebildet werden müssen. Dies geht natürlich zu Lasten der ebenfalls eingeschränkten und daher kostbaren Ressource Speicherplatz. Der durch die RISC Architektur eingesparte Energie- und teilweise auch physikalische Platzverbrauch des Prozessors wiegt den etwas erhöhten Speicherbedarf der RISC Programme in der Regel auf.

C. Czernohous, *Pervasive Linux*, X.systems.press,
DOI 10.1007/978-3-540-68426-8_2, © Springer-Verlag Berlin Heidelberg 2012

Die für Pervasive Computing momentan am häufigsten eingesetzten Prozessoren basieren auf den RISC Architekturen PowerPC, Advanced RISC Machines (ARM) und MIPS, sowie der CISC-Architektur M68000.

2.2 Speicher

Pervasive Computing stellt auch an die einsetzbaren Speicheralternativen besondere Anforderungen.

2.2.1 Arbeitsspeicher

Arbeitsspeicher ist die Systemkomponente, aus der der Prozessor Daten und Befehle zu deren Verarbeitung erhält und in den die Ergebnisse nach Ausführung der Befehle abgelegt werden.

Der Hauptspeicher besteht in den meisten Fällen aus *Dynamic Random Access Memory* (*DRAM*) Speicherbausteinen. Die Speicherverwaltungseinheit des Prozessors kann auf den DRAM Speicherbereich beliebig lesend und schreibend zugreifen.

DRAM ist sehr schnell und eignet sich daher für den Einsatz als Hauptspeicherkomponente, die für eine effiziente Kommunikation mit der Prozessoreinheit über schnelle Antwortzeiten verfügen muss. DRAM Speicherbausteine benötigen eine regelmäßige Stromzufuhr in kurzen Intervallen, um den gespeicherten Datenbestand aufrechtzuerhalten. Sobald die Energiezufuhr unterbrochen wird, gehen die gespeicherten Daten verloren.

2.2.2 Permanenter Speicher

Um Daten über einen längeren Zeitraum und mögliche Unterbrechungen der Energiezufuhr hinweg zu speichern, werden permanente, so genannte nicht flüchtige Speicherformen benötigt. Für den Einsatz in mobilen Geräten haben sich diverse Varianten sogenannten Flash-Speichers etabliert. Aber auch besonders kleine, stromsparende und robuste Festplatten kommen zum Einsatz.

Um den Zugriff auf Speicher, insbesondere Flash-Speicher zu vereinheitlichen entwickelt das Projekt *Memory Technology Device* (*MTD*) *Subsystem for Linux* eine Abstraktionsschicht, so dass Gerätetreiber sich auf die Basiszugriffsfunktionen beschränken können. Im Internet ist das Projekt zu finden unter http://www.linux-mtd.infradead.org.

2.2.2.1 Flash

Flash Speicher ist die am häufigsten anzutreffende Speicherart für die dauerhafte Ablage von Daten in mobilen Geräten. Bei Flash handelt es sich um

Speicherbausteine, die ihren Inhalt auch ohne permanente Stromversorgung behalten. Sie beinhalten keine mechanischen Komponenten und verfügen über eine Zugriffsgeschwindigkeit, die mit gängigen Festplatten vergleichbar ist.

Die Speicherbereiche von Flash Speicher können nur begrenzt oft beschrieben werden, in der Regel einige tausend Mal. Das macht sie zu einer besonders wertvollen Komponente, die mit Bedacht eingesetzt werden muss. Insbesondere ist darauf zu achten, dass die Speichereinheiten, in die der verfügbare Gesamtspeicher eingeteilt ist, möglichst gleichmäßig beschrieben werden, um die Lebensdauer zu maximieren.

Aus diesem Grund ist es wichtig, dass die für Flash Speicher verwendeten Dateisysteme über sogenanntes *Wear Leveling* verfügen, das eine Verschleiss-Gleichverteilung der Schreibzugriffe über den vorhandenen Speicherbereich sicherstellt. Speziell für diesen Einsatzzweck wurden geeignete Dateisysteme entwickelt, die in in Abschn. 3.4 vorgestellt werden.

Flash Speicher wird in so genannte Löschblöcke (engl.: *Erase Blocks*) aufgeteilt, die üblicherweise deutlich grösser sind als die von Festplatten bekannte Blockgrösse. Eine Besonderheit von Flash Speicher ist es, dass der zu beschreibende Bereich zuerst gelöscht werden muss, bevor eine Schreiboperation möglich ist. Das Löschen eines Blocks erfolgt, indem allen Bits des Blocks der Wert Eins zugewiesen wird. Der technische Hintergrund dafür ist, dass die einzig mögliche Schreiboperation bei Flash-Speicher ist, den Wert einzelner Bits von eins auf null zu setzen. Das bedeutet, dass ein Block, damit er neu beschrieben werden kann, zuerst gelöscht werden muss, sobald mindestens ein Bit des Blocks, das nach der Schreiboperation den Wert eins annehmen soll, vor der Schreiboperation eine Null enthält.

Zur Zeit gibt es zwei Typen von Flash Speichern, die unterschiedliche technische und ökonomische Eigenschaften aufweisen, so genannte *NOR*- und *NAND* Speicherbausteine.

NAND Speicher ist sequenziell organisiert, d.h. ein Speicherblock wird nach dem anderen gelesen, ohne direkt auf einen bestimmten Speicherblock zugreifen zu können. Die Schreiboperationen für NAND Speicher sind sehr schnell. Im Gegensatz dazu kann auf NOR Speicher wahlfrei (engl.: random access) zugegriffen werden, allerdings sind Schreiboperationen im Vergleich zu NAND Speicher relativ langsam. NOR Speicherchips sind physikalisch grösser als entsprechender NAND Speicher und können nicht so oft wiederbeschrieben werden.

CompactFlash

Bei Speicher vom Typ *CompactFlash* (*CF*) handelt es sich um Flash-Speicher, der bereits mit einer Steuereinheit versehen ist und als Einheit mit dieser geliefert wird. Diese Steuereinheit sorgt unter anderem für das bereits angesprochene Wear Leveling, also die möglichst gleichmäßige Verteilung der Speicherzugriffe auf den bereitgestellen Gesamtspeicher der Einheit. Die entsprechenden Spezifikationen werden von der *CompactFlash Association* (*CFA*) entwickelt und auf der Seite http://www.compactflash.org zur Verfügung gestellt.

CF-Speicher wird häufig in Form von Speicherkarten angeboten, die etwa die Größe einer halben Kreditkarte haben und z. B. auch zur Ablage von Bilddaten in digitalen Fotoapparaten zum Einsatz kommen. Ursprünglich wurde für CF-Karten NOR Flash-Speicher verwendet, inzwischen kommt aber NAND Flash zum Einsatz.

Üblicherweise emulieren CF-Karten eine Festplatte, so dass sie vom Betriebssystem als Festplatte erkannt und verwaltet werden.

Im Gegensatz zu dem bereits beschriebenen Flash Speicher sollte für CF-Speicher kein Dateisystem eingesetzt werden, das für herkömmlichen Flash-Speicher entwickelt wurde, da die gleichmäßige Verteilung der Schreibzugriffe bereits durch die vorhandene Steuereinheit sichergestellt wird. Derartige Dateisysteme verschlechtern normalerweise die Zugriffszeiten auf die Speichereinheit, bieten aber keinen zusätzlichen Nutzen.

MultiMediaCard

Speicherkarten vom Typ *MultiMediaCard* (MMC) basierten von Anfang an auf NAND Flash-Speicher und sind daher deutlich kleiner als CF-Speicherkarten. Durch die geringe physikalische Grösse und die schnellen Zugriffszeiten kommen MMC-Karten häufig in Geräten des Pervasive Computing zum Einsatz.

Secure Digital

Der MMC-Standard wurde weiterentwickelt und mit Sicherheitsfunktionen versehen, die es z. B. Medienunternehmen erlauben, rechtlich geschützte Inhalte wie Musikstücke mit einem Zugriffsschutz zu versehen und auf Flash-Speicherkarten auszuliefern. Daraus entstand der *Secure Digital* (SD)-Standard für Speicherkarten.

Mit dem Standard *microSDmicroSD* wurde ein Standard auf Basis von SD-Karten mit sehr kleinen physikalischen Abmessungen geschaffen. microSD-Karten haben die gleichen Eigenschaften wie herkömmliche SD-Karten, sind aber nur knapp eineinhalb Quadratzentimeter gross und daher hervorragend für den Einsatz in besonders kleinen Geräten geeignet.

2.2.2.2 Festplatte

Die aus der Welt der Personal Computer allgegenwärtigen und bewährten Festplatten werden in zunehmendem Maße auch in Geräten des Pervasive Computing eingesetzt. Inzwischen gibt es Festplatten mit geringem Energieverbrauch und mehreren Gigabyte Speicherkapazität in der Größe einer halben Kreditkarte. Ein Beispiel ist das von der Firma Hitachi hergestellte Microdrive.

Festplatten haben im Vergleich zu den bereits beschriebenen Flash Speicherarten den Vorteil einer wesentlich höheren Anzahl möglicher Schreibzugriffe und damit eine in der Regel höhere Lebensdauer bei vergleichbaren Zugriffszeiten. Allerdings werden Festplatten immer mehr durch die diversen Alternativlösungen mit Flash-Speicher abgelöst.

2.3 Eingabe und Ausgabe

Eingabemöglichkeiten im Pervasive Computing reichen von externen Tastaturen über eingeblendete Tastaturabbilder auf berührungsempfindlichen Bildschirmen bis zur Handschrifterkennung in dafür vorgesehenen Eingabebereichen. Weitere Quellen sind Kameras, Mikrofone, Barcode-Scanner, Bewegungssensoren und Module zur Bestimmung der aktuellen Position, um nur einige Beispiele zu nennen.

Für die Ausgabe gibt es Bildschirme unterschiedlichster Grössen und Auflösungen. Lautsprecher gehören in modernen mobilen Geräten oft zur Ausstattung.

Über eine beachtliche Anzahl möglicher Netzwerkverbindungen wird mit dem Umfeld kommuniziert.

Diesen Variantenreichtum gilt es ähnlich wie bei herkömmlichen PC-Anwendungen zu berücksichtigen.

2.4 Energieversorgung

Das Thema Energieversorgung spielt beim Pervasive Computing eine besondere Rolle. Viele Geräte in diesem Umfeld sind mobil und daher auf eine stromnetzunabhängige Energieversorgung angewiesen. Aus diesem Grund werden in den meisten Fällen für diesen Einsatzzweck optimierte Prozessoren eingesetzt. Moderne Prozessoren können in verschiedenen Betriebsarten abhängig vom momentanen Einsatzzweck eingesetzt werden. Für multimediale Anwendungen wird z. B. mehr Rechenleistung benötigt und daher in einen verbrauchsintensiveren Zustand gewechselt. Diese Tatsache wirkt sich auch auf die für diese Geräte entwickelte Software aus. Beispielsweise muss damit gerechnet werden, dass die Stromversorgung, und damit auch der Betrieb des Gerätes, plötzlich abbricht. Damit in diesem Fall der Datenverlust minimiert wird, müssen entsprechende Vorkehrungen getroffen werden (Abschn. 1.3.2).

Ausserdem trägt eine ressourcenoptimierte Herangehensweise zur Softwareentwicklung dazu bei, dass die vorhandenen Energiereserven geschont werden und damit die Laufzeit von Systemen, die nicht über eine permanente Energieversorgung versorgt werden, verlängert wird.

Aus diesem Grund bieten Systeme nicht nur aus dem Bereich des Pervasive Computing zahlreiche Möglichkeiten, den Energieverbrauch zu optimieren.

Kapitel 3
Software

Die wichtigsten Komponenten jedes Linux Systems sind der Boot Loader, der Linux Kernel, sowie der Inhalt des Root Dateisystems, wobei der Boot Loader streng genommen nicht zum eigentlichen Linux System gehört und betriebssystemunabhängig ist. Die genannten Komponenten werden in diesem Kapitel vorgestellt.

Da damit zwar ein vollständiges Linux System beschrieben ist, dieses ohne Anwendungssoftware aber verhältnismässig nutzlos ist, wird auch auf Alternativen zum Dialog mit dem Benutzer eingegangen. Die unterschiedlichen Benutzerschnittstellen sind Voraussetzung für die Integration von Anwendungsprogrammen in das System. Darüber hinaus werden momentan verfügbare Distributionen vorgestellt und auf Möglichkeiten zum Datenaustausch eingegangen.

Zur Ablage von Programmen und Nutzdaten spielen Dateisysteme eine zentrale Rolle. Da sich die für Pervasive Computing verwendete Hardware oft stark von der im Desktop Umfeld verbreiteten unterscheidet, existieren einige spezialisierte Dateisystemtypen, deren Vor- und Nachteile gegenübergestellt werden.

3.1 Boot Loader

Nach dem Einschalten lädt der Rechner einen Programmcode von einem definierten Ort in den Hauptspeicher und die CPU-Register und führt diesen aus. Diese Software wird als Boot Loader bezeichnet und ist dafür verantwortlich, das Betriebssystem zu starten. Die meisten für Linux einsetzbaren Boot Loader erlauben darüber hinaus über einfache Dialoge die Auswahl unterschiedlicher Betriebssysteme oder Kernelkonfigurationen, sofern mehr als ein System installiert ist. Prinzipiell ist ein Boot Loader unabhängig von einem bestimmten Betriebssystem, jedoch wurden die meisten Boot Loader zusammen mit einem konkreten Betriebssystem entwickelt und bieten daher für dieses Betriebssystem die beste Unterstützung. Das gilt auch für die hier vorgestellten Linux-nahen Boot Loader.

Im Gegensatz zu Desktop PCs, bei denen eine spezielle Firmware, das so genannte *Basic Input Output System* (*BIOS*), die Hardwarebestandteile initialisiert, besitzen Geräte aus dem Pervasive Computing Umfeld meistens kein BIOS. Daher muss der Boot Loader die Hardware initialisieren, ggf. den Betriebssystem-Kernel in den Arbeitsspeicher laden und den Kernel mit möglichen Parametern starten. Die

C. Czernohous, *Pervasive Linux*, X.systems.press,
DOI 10.1007/978-3-540-68426-8_3, © Springer-Verlag Berlin Heidelberg 2012

Möglichkeit, über eine Eingabezeile des Bootloaders Parameter an den Kernel zu übergeben, wird Kernel-Kommandozeile (engl.: Kernel Command Line) genannt, obwohl es sich dabei nicht um eine Befehlszeile im Sinne von Linux handelt.

Ausserdem bieten viele Boot Loader die Möglichkeit, das Betriebssystem und zusätzliche Programme auf das Zielgerät zu übertragen. Üblicherweise wird ein zuvor erstelltes Speicherabbild mit Hilfe eines unterstützten Kommunikationsprotokolls wie FTP oder ZMODEM an den Boot Loader geschickt und von diesem in den Flash-Speicher des Zielgerätes geschrieben.

Der Boot Loader selbst wird auf unterschiedliche Weise installiert und das konkrete Vorgehen hängt von der verwendeten Ziel-Hardware ab. Idealerweise ist bereits vom Hardware-Hersteller ein Boot Loader installiert, der direkt verwendet werden kann oder es zumindest erlaubt, eine aktuelle Version oder sogar einen anderen Boot Loader zu installieren. Falls sich keinerlei Software auf dem Zielgerät befindet, muss der Boot Loader mit anderen Hilfmitteln, wie externen Flash-Programmiergeräten, auf das Zielgerät gebracht werden. Die Beschreibung der dazu notwendigen Schritte ist nicht Bestandteil dieses Buches und sollte beim Hardware-Hersteller erfragt werden.

Mittlerweile gibt es eine große Anzahl an Boot Loadern für alle gängigen Hardware-Architekturen. Die folgenden Kapitel stellen nur eine Auswahl der verbreitetsten Boot Loader dar. Teilweise können auf einem Gerät unterschiedliche Boot Loader alternativ installiert werden. Daher ist es hilfreich, mit den am häufigsten anzutreffenden Boot Loadern vertraut zu sein.

3.1.1 Das U-Boot – Universal Boot Loader

Das U-Boot ist der wohl verbreitetste Open Source Boot Loader für eingebettete Linux-Systeme (http://sourceforge.net/projects/u-boot/). „Das U-Boot“ ist das Nachfolgeprojekt von *PPCBoot*, einem flexiblen und vielseitig einsetzbaren Boot Loader für Prozessoren der Power-Architektur. Die Offenheit und Flexibilität von PPCBoot führte dazu, dass der Boot Loader für weitere Prozessorachitekturen portiert wurde. Inzwischen unterstützt er alle gängigen CPU-Architekturtypen wie ARM, XScale, MIPS, x86 und natürlich PowerPC.

Für „Das U-Boot“ steht auch ein Satz Entwicklungswerkzeuge in Form des Embedded Linux Development Kit (ELDK) sowie umfangreiche Benutzerdokumentation zur Verfügung.

3.1.2 ARM Boot Loader

Die Entwicklung des ARM Boot Loader (http://www.handhelds.org) wird von der Firma Hewlett-Packard unterstützt, da dieser speziell für den Einsatz auf HP iPaq PDAs optimiert ist. Er unterstützt das Starten diverser Betriebssysteme direkt aus dem Flash-Speicher oder von einer CompactFlash Speicherkarte. Der Linux Kernel wird vom gewählten Speichermedium (Flash oder CF) in den Hauptspeicher entpackt und dann ausgeführt.

3.1.3 Linux Loader (LILO)

Für die x86 PC-Plattform war LILO lange Zeit der Standard, was Boot Loader betrifft. LILO kann sehr gut mit den vielen Einschränkungen der PC-Architektur umgehen, die sich durch die Weiterentwicklung der Hardware-Bestandteile unter Berücksichtigung der Kompatibilität mit älteren PC-Systemen ergeben haben. LILO kann auch praktisch alle gängigen PC-Betriebssysteme starten.

Der grösste Vorteil von LILO, nämlich die Anpassungsfähigkeit an unterschiedlichste Boot-Konzepte im PC Umfeld ist umgekehrt auch der grösste Nachteil. LILO ist auf die x86 Plattform beschränkt. Ausserdem benötigt LILO für den Boot-Vorgang ein BIOS, von dem es die Partitionsinformationen der Festplatten im System erhält.

LILO kann zum Boot-Zeitpunkt keine Dateisysteme lesen. Die Information, wo auf einer Festplatte sich der zu startende Linux Kernel befindet, wird zum Installationszeitpunkt ausgelesen und der LILO-Installation übergeben. Das hat zur Folge, dass LILO jedes Mal neu installiert werden muss, wenn die Konfiguration des Linux Kernels geändert wird.

Aus den genannten Nachteilen ergibt sich, dass LILO für die Verwendung im Pervasive Computing eher ungeeignet ist, es sei denn die Zielplattform basiert auf der x86 Architektur und verfügt über ein BIOS. Aufgrund der großen Verbreitung von LILO und der gemeinsamen Historie zusammen mit Linux selbst, darf dieser Boot Loader nicht unerwähnt bleiben.

3.1.4 Grand Unified Boot Loader (GRUB)

GRUB (http://www.gnu.org/software/grub) ist wie LILO ein Boot Loader für Personal Computer und nicht speziell für Pervasive Computing Geräte entwickelt worden. Jedoch hat die Weiterentwicklung von GRUB in Form von GRUB 2 u.a. zum Ziel, die Portierbarkeit auf andere Plattformen zu erleichtern. Damit wird GRUB 2 zunehmend auch für den Einsatz auf Pervasive Computing Geräten interessant.

Der Vorteil von GRUB im Vergleich zu LILO ist seine grössere Flexibilität bezüglich Installation und Konfiguration. So kann GRUB beispielsweise direkt eine Reihe der wichtigsten Dateisysteme lesen und ist damit in der Lage, z. B. einen Linux-Kernel anhand des Dateinamens zu starten, anstatt den Kernel über eine physikalische Adresse aus dem nicht-flüchtigen Speicher zu laden. Ausserdem wird GRUB über eine Datei konfiguriert und muss nicht nach jeder Änderung der Konfiguration neu installiert werden, wie das bei LILO der Fall ist.

3.1.5 blob

blob (http://sourceforge.net/projects/blob) ist als Boot Loader aus dem Universitätsprojekt Linux Advanced Radio Terminal (LART) hervorgegangen (http://www.lart.tudelft.nl). LART ist ein an der Universität Delft entwickeltes Embedded Linux

System. Dieser Boot Loader wird auf StrongARM Prozessoren eingesetzt und wurde bewusst einfach gehalten, um hohe Effizienz bei geringem Ressourcenverbrauch zu erreichen.

3.1.6 *Micromonitor* (*uMon*)

Micromonitor (http://www.microcross.com/html/micromonitor.html) ist ein Boot Loader, der von Ed Sutter bei der Firma Lucent Technologies entwickelt und als Open Source freigegeben wurde.

Dieser Boot Loader ist für zahlreiche Hardwareplattformen verfügbar, u.a. ARM, PowerPC, XScale und MIPS und kann ausser Linux fast jedes Betriebssystem oder spezielle Anwendungen, die ohne Betriebssystem auskommen, starten.

Micromonitor besitzt ein eigenes Dateisystem mit dem Namen Trivial File System (TFS) und wird über ASCII Dateien konfiguriert. Der Startvorgang kann mit Hilfe einer an die Programmiersprache BASIC angelehnten Skriptsprache konfiguriert und gesteuert werden. Nach dem Start der Anwendung besteht die Möglichkeit, über ein API auf Micromonitor zuzugreifen, beispielsweise um Umgebungsvariablen auszulesen, die während des Startvorgangs gesetzt wurden. Zum Beispiel kann von Micromonitor die TCP/IP Konfiguration der Netzwerkschnittstelle initialisiert werden und diese Information dann von der gestarteten Anwendung (z. B. Linux) ausgelesen und übernommen werden.

Mit Micromonitor kann die Anwendung entweder aus dem auf dem Zielgerät vorhandenen Speicher oder über ein Netzwerk gestartet werden. Für den Start der Anwendung aus dem Netzwerk unterstützt Micromonitor das Trivial File Transfer Protocol (TFTP), mit dem der Programmcode in den Speicher des Zielgeräts übertragen wird.

3.1.7 *RedBoot*

Der von der Firma RedHat im Rahmen des Projektes eCos entwickelte Boot Loader *RedBoot* (http://sources.redhat.com/redboot/) ist ein sehr flexibler und komplexer Boot Loader für das Pervasive Computing, dessen Funktionalität über die Aufgaben eines Boot Loaders hinausgeht. RedBoot unterstützt eine grosse Anzahl Prozessorarchitekturen und -typen.

3.1.8 *colilo*

Der *colilo* Boot Loader wurde für Motorola Coldfire Prozessoren entwickelt, die wiederum auf der M68000 Prozessorarchitektur basieren. Er findet deshalb hier Erwähnung, da er zum Starten von ucLinux eingesetzt wird, eine Version von Linux für Prozessoren ohne Speicherverwaltungseinheit (engl. Memory Management Unit – MMU).

3.1.9 Qi

Qi ist ein minimalistischer Boot Loader, dessen Funktionalität sich auf den eigentlichen Boot-Vorgang konzentriert. Dabei wird besonderer Wert auf die Unterstützung von SD und SDHC Karten gelegt. Hervorgegangen ist der Boot Loader aus dem Openmoko-Projekt. Zu finden ist Qi unter http://wiki.openmoko.org/wiki/Qi.

3.2 Linux Kernel

Der Betriebssystemkern, der so genannte Linux Kernel, stellt die zentrale Komponente jedes Linuxsystems dar. Der Kernel wird vom Boot Loader aufgerufen und damit wird das System gestartet. Im Kernel sind die Betriebssystemfunktionen wie Speicher- und Prozessverwaltung oder Steuerung der Ein- und Ausgabe implementiert. Der Kernel steuert den Systemstart, den so genannten Bootprozess, indem die benötigten Gerätetreiber für die vorhandene Hardware geladen und initialisiert werden, konfigurierte Systemdienste (engl.: Services) gestartet werden, Umgebungsvariablen des Systems gesetzt werden und das System somit in einen definierten Zustand versetzt wird.

Die zentrale Webseite für den Quellcode des Linux Kernels ist http://www.kernel.org. Hier finden sich auch Verweise auf Webseiten der Projekte, die den Linux Kernel von der x86 Architektur auf zahlreiche andere Hardwareplattformen portieren.

Der Funktionsumfang des Kernels und damit auch die Grösse der resultierenden Binärdatei kann vor dem Übersetzen umfangreich konfiguriert und an die Eigenschaften der Zielplattform angepasst werden. Das ist gerade für das Pervasive Computing wichtig, da somit der Speicherbedarf des Kernel minimiert und kostbarer Speicherplatz eingespart werden kann.

3.2.1 Kernel Module

Zahlreiche Funktionalitäten wie z. B. Gerätetreiber lassen sich wahlweise als fester Bestandteil des Kernels oder als so genanntes Kernel Modul übersetzen. Kernel Module werden zur Laufzeit erst dann in den Arbeitsspeicher geladen, wenn konkreter Bedarf für die Funktionalität dieses Moduls besteht und werden nach längerer Nichtbenutzung auch wieder aus dem Speicher entfernt. Das hat den Vorteil, dass die Grösse des durchschnittlich durch den Kernel belegten Hauptspeichers sich verringert. Im Gegensatz dazu sollten häufig genutzte Funktionalitäten fest in den Kernel übersetzt werden, da ein zu häufiges Laden und wieder Entladen eines Kernel Moduls unnötig Rechenzeit und Ressourcen verbraucht. Die optimale Konfiguration besteht in den meisten Fällen aus einer Mischung aus Kernel Modulen und festen Kernelbestandteilen und hängt von Einsatzzweck und Nutzungsverhalten des Zielsystems ab.

3.2.2 Framebuffer

Das so genannte *Framebuffer-Gerät* abstrahiert den Zugriff auf physikalische Grafikgeräte. Es stellt eine Schnittstelle bereit, über die Anwendungsprogramme grafische Ausgaben auf einheitliche Weise durchführen können. Dieses Konzept der Entkoppelung von hardwareabhängigem Gerätetreiber und Anwendungsprogramm birgt gerade für das Pervasive Computing zwei entscheidende Vorteile. Zum einen funktioniert für alle Grafikgeräte, die Framebuffer unterstützen, d. h. für die ein Framebuffer Gerätetreiber existiert, die Darstellung von Grafik identisch, ohne dass sich der Anwendungsentwickler mit Details der eingesetzten Hardware bzw. mit der Verwendung des Gerätetreibers auskennen muss. Zum anderen gilt diese Aussage ebenso für die Kompatiblität über unterschiedliche Hardwarearchitekturen hinweg. Dadurch ist der Teil einer Software, der für die Grafikausgabe verantwortlich ist und dafür das Framebuffer-Gerät verwendet, sehr leicht portierbar, sowohl für unterschiedliche Grafikgeräte der gleichen Hardwareplattform, als auch über unterschiedliche Plattformen hinweg.

Die Framebuffer-Implementierung unter Linux ist aufgrund der ressourcensparenden Implementierung gerade für das Pervasive Computing interessant.

Seit der Kernelversion 2.2 ist die Unterstützung von Framebuffer-Geräten auch auf der Intel x86 Architektur Bestandteil des Linux-Kernels.

3.2.3 uClinux – virtuelle Speicherverwaltung

Das Projekt *uClinux* hat es sich zur Aufgabe gemacht, Linux auf Hardware-Plattformen zu portieren, die über keine Speicherverwaltungseinheit (engl.: Memory Management Unit – MMU) verfügen. Prozessoren für Personal Computer beinhalten normalerweise eine MMU, die den Zugriff auf den Hauptspeicher kontrolliert und verwaltet, sie ist also Bestandteil der Hardware. Um Ressourcen und Platz einzusparen fehlt diese Einheit in manchen Prozessoren, die im Pervasive Computing eingesetzt werden. Für Betriebssysteme wie Linux, die das gleichzeitige Ausführen von Prozessen ermöglichen, ist eine MMU erfoderlich, da sich mehrere Programme den verfügbaren Hauptspeicher teilen und dessen Verwaltung nicht dem jeweiligen Programm überlassen werden kann. Fehlt also eine MMU muss die entsprechende Funktionalität als Bestandteil des Betriebssystems implementiert werden. Inzwischen ist uClinux in die offiziellen Quellen des Linux Kernels integriert. Im Internet ist das Projekt unter http://www.uclinux.org zu finden.

3.3 Gerätetreiber

Die Schnittstelle zwischen Hardware und Betriebssystem bilden so genannte. Unter Linux gibt es zwei Typen von Gerätetreibern, die je nachdem, ob der Zugriff auf die Hardware zeichenorientiert (engl.: Character Device) oder in gösseren Speicherblöcken (engl.: Block Device) erfolgt. Inzwischen gibt es noch eine dritte Kategorie

in Form von *Memory Technology Device* (*MTD*), die insbesondere für den Zugriff auf Flash-Speicher und Hauptspeicherbereiche entwickelt wurde. Die Idee hinter dem Konzept von Gerätetreibern unter UNIX und Linux ist es, den Zugriff auf die entsprechende Hardware wie das Lesen und Schreiben von Dateien zu behandeln.

3.3.1 Character Device

Zeichenorientierte Geräte (*engl.: Character Devices*) stellen Daten in Form eines Zeichenstroms zur Verfügung, der Zeichen für Zeichen ausgelesen werden kann. Es ist dabei nicht möglich, an eine bestimmte Stelle im Zeichenstrom zu springen, um die aktuelle Leseposition zu verändern. Auch ist es bei diesen Geräten nicht möglich, die Grösse der Datenmenge von vornherein zu bestimmen.

3.3.2 Block Device

Bei *blockorientierten Geräten* (*engl.: Block Device*) werden die bereitgestellten Daten in gleich grosse Blöcke unterteilt und entsprechend blockweise gelesen. Die Gesamtdatenmenge ist bekannt und kann entsprechend abgefragt werden. Ausserdem ist es möglich, wahlfrei auf die Datenblöcke zuzugreifen.

3.3.3 Memory Technology Device (MTD)

Die Eigenschaften von Flash Speicher (Abschn. 2.2.2.1) passen weder in die Kategorie der block-, noch in die der zeichenorientierten Geräte. Aus diesem Grund wurde ein neuer Gerätetyp eingeführt in Form von *Memory Technology Device* (*MTD*) Geräten. Das entsprechende Projekt zur Entwicklung des Subsystems befindet sich auf der Seite http://www.linux-mtd.infradead.org/. Ziel ist es, über eine Software-Schicht von der Flash-Hardware zu abstrahieren.

Für die Verwaltung von Flash Speicher mit MTD gibt es ein Paket unterstützender Programme, die *MTD-Utils*.

3.3.3.1 Unsorted Block Images

Mit *Unsorted Block Images* (*UBI*) bietet das MTD Projekt eine Abstraktionsschicht an, die die Verwaltung des physikalischen Flash-Bereiches übernimmt. Dabei wird der vorhandene Speicherbereich auf logische Speichermedien (*engl.: Volumes*) abgebildet. Diese logischen Einheiten sind entweder *statisch*, d.h. es kann auf sie nur lesend zugegriffen werden, oder *dynamisch*, so dass auch Schreiboperationen möglich sind.

UBI-Einheiten können bei Bedarf erzeugt, gelöscht oder in der Grösse verändert werden. Die UBI-Schicht übernimmt das *Wear Levelling*, und sorgt damit für eine

Gleichverteilung der Schreibzugriffe, das Aussortieren fehlerhafter Bereiche und den gleichmässigen Verschleiss des verwalteten Speichers.

3.3.3.2 Flash Translation Layer

Über ein so genanntes *Flash Translation Layer* (*FTL*) kann auf ein MTD Gerät wie auf ein blockorientiertes Gerät zugegriffen werden. Diese zusätzliche Softwareschicht emuliert ein blockorientiertes Gerät für jedes vorhandene MTD Gerät und stellt diese als Gerätedateien `/dev/mtdblockN` im System zur Verfügung, wobei `N` aufsteigend numeriert wird.

Damit ist es möglich, gängige Dateisysteme, die für blockorientierte Geräte entwickelt wurden auf MTD Geräten einzusetzen. Allerdings gehen mit FTL die Vorteile und spezifischen Eigenschaften von Flash und MTD verloren bzw. bleiben ungenutzt.

3.4 Dateisystemtypen

Normalerweise greifen Anwendungsprogramme nicht direkt auf den Gerätetreiber eines Datenträgers zu. Dateisysteme stellen die Schnittstelle zwischen Anwendungsprogramm und Gerätetreiber für einen bestimmten Datenträger dar. Sie steuern, wie Daten auf einem Datenträger organisiert werden und verwalten die Dateistruktur.

Das Betriebssystem nutzt ein Dateisystem, um Daten auf einen Datenträger zur schreiben und von ihm zu lesen. Mit jedem Schreibzugriff auf den Datenträger ändert sich die Informationen über die Organisation und den Inhalt des Datenträgers, die so genannten Metadaten. Geht diese Information verloren, sind auch die Nutzdaten des Datenträgers nicht mehr lesbar und damit wertlos. Aus Performanzgründen werden die Metadaten meistens nicht sofort auf dem Datenträger abgelegt, sondern in einem schnellen Zwischenspeicher (engl.: *Cache*) gehalten und in definierten Abständen persistent gespeichert. Fällt das System zwischen einer Änderung der Nutzdaten und der persistenten Ablage der Metadaten aus, kommt es zu Datenverlust, da die Information darüber, wo und wie die Nutzdaten auf dem Datenträger abgelegt sind, verlorgen gegangen ist.

Nach dem Wiederanlauf des Systems muss nach einem Systemausfall der gesamte Datenträger überprüft werden, um die Metadaten konsistent wiederherzustellen. Der für die Rekonstruktion benötigte Zeitaufwand wächst mit der Grösse des zu verwaltenden Speicherbereichs.

Journallierende Dateisysteme (engl.: *Journaling File Systems*) beugen einer möglichen Inkonsistenz der Metadaten vor, indem alle die Dateistruktur verändernden Zugriffe auf den Datenträger seit der letzten persistenten Sicherung der Metadaten in einem gesonderten Speicherbereich des Datenträgers, dem so genannten Journal, einer Art Logbuch, mitprotokolliert werden, bevor die Veränderung der Dateistruktur durchgeführt wird. Mit Hilfe dieses Protokolls ist es möglich, nach einem Systemausfall sehr schnell die Metadaten konsistent wiederherzustellen und den

Datenverlust zu minimieren. Das ständige persistente Mitprotokollieren der Metadaten führt im Vergleich zu herkömmlichen Dateisystemen zu einer deutlich höheren Anzahl an Schreibzugriffen, was sich negativ auf die Lebensdauer von Flash-Speicher auswirkt. Aus diesem Grund entstanden eine Reihe für Flash-Speicher optimierte journallierende Dateisysteme.

3.4.1 *Journaling Flash File System (JFFS)*

Das Dateisystem *JFFS* wurde von der schwedischen Firma Axis Communications speziell für Flash Speicher vom Typ NOR entwickelt (http://developer.axis.com/software/jffs/). Dieses journallierende Dateisystem unterstützt die Version 2.0 des Linux Kernels. JFFS löscht und schreibt Speicherblöcke relativ häufig, was zu einem schnellen Verschleiss der Flash-Hardware führt.

3.4.2 *Journaling Flash File System, Version 2 (JFFS2)*

Auf Basis von JFFS entwickelte die Firma Red Hat eine neue Version des Dateisystems, das Journaling Flash File System, Version 2 (http://sourceware.org/jffs2/). Seit der Linux Kernel Version 2.4.10 ist JFFS2 Bestandteil des Linux Quellcode. Inzwischen unterstützt JFFS2 auch Flash Speicher vom Typ NAND.

Beim Einhängen eines JFFS2 Dateisystems wird dieses gelesen und der Index für das Journal im Hauptspeicher aufgebaut. Ausserdem kann der Inhalt eines JFFS2 Dateisystems zusätzlich komprimiert werden.

3.4.3 *Unsorted Block Images File System (UBIFS)*

Mit *UBIFS* steht ein Dateisystem für Flash Speicher zur Verfügung, das auf UBI Speichermedien (Abschn. 3.3.3) eingesetzt wird. Im Gegensatz zu JFFS2 legt UBIFS den Index des Journals nicht im Hauptspeicher, sondern ebenfalls im Flash Speicher ab. Das hat den Vorteil, dass der Index nicht jedes Mal beim Einhängen eines Verzeichnisses neu aufgebaut werden muss und dass die Grösse des Dateisystems nicht von der Grösse des Hauptspeichers abhängt. Entwickelt wird UBIFS von der Firma Nokia in Zusammenarbeit mit der Universität Szeged.

3.4.4 *LogFS*

Das Projekt *LogFS* (http://logfs.org) hat es sich zum Ziel gesetzt, die Nachteile, insbesondere sinkende Zugriffsgeschwindigkeit von JFFS2 bei gösseren Flash Speichern, zu beheben. LogFS implementiert ein journallierendes Dateisystem.

3.4.5 Yet Another Flash File System (YAFFS)

Aus der Bezeichnung dieses Dateisystems lässt sich herauslesen, dass es nicht das erste Dateisystem für Flash Speicher ist, übersetzt heisst es in etwa *Noch ein weiteres Flash Dateisystem*. YAFFS (http://www.yaffs.net/) wurde für Flash Speicher vom Typ NAND entwickelt, um auf die spezifischen Eigenschaften der NAND Technik einzugehen, unterstützt aber sowohl NAND als auch XOR Flash Speicher. YAFFS ist wie JFFS und JFFS2 ein journallierendes Dateisystem.

Mit *YAFFS2* gibt es eine Parallelentwicklung für NAND Speicher, die mit optimierten Algorithmen eine höhere Geschwindigkeit und weniger Hauptspeicherverbrauch erzielt. Zudem werden die Möglichkeiten neuerer Hardware ausgenutzt.

3.4.6 Squashfs

Squashfs (http://squashfs.sourceforge.net) bietet effiziente Komprimierung und ausschliesslich Lesezugriff auf Dateien. Damit eignet es sich sehr gut für bestimmte Einsatzbereiche im Pervasive Computing. Z. B. können die für den Bootvorgang benötigten Dateien des Root-Dateisystems platzsparend in einem Squashfs Dateisystem abgelegt werden. Dieses Prinzip wird auch zunehmend von so genannten *Live CDs* genutzt, wobei eine Linuxdistribution von einer CD ROM oder DVD direkt startfähig ist.

3.4.7 (cramfs)

cramfs (http://sourceforge.net/projects/cramfs/) ist ein einfaches komprimierendes Dateisystem für den Lesezugriff. Die Implementierung benötigt selbst nur sehr wenig Speicherplatz und erlaubt wahlfreien Zugriff auf den Speicherbereich. Die Dekomprimierung der zu lesenden Daten erfolgt beim Lesen automatisch.

3.4.8 Second Extended File System (EXT2), EXT3 und EXT4

Das Dateisystem *EXT2* war lange Zeit das Standarddateisystem für Linux Systeme im Desktop-Umfeld.

In der von der Firma Red Hat vorangetriebenen Weiterentwicklung unter dem Namen *EXT3* wurde es um Journallierungsfähigkeit erweitert und hat EXT2 als den eingesetzten Standard abgelöst. EXT3 ist abwärtskompatibel zu EXT2, d.h. dass mit EXT3 formatierte Partitionen auch als EXT2 lesbar sind. Seit der Kernel Version 2.4.15 ist EXT3 Bestandteil des Linux Quellcode.

Inzwischen gibt es eine erneute Weiterentwicklung des Dateisystems unter dem Namen *EXT4*. Hauptverbesserungspunkt der neuen Generation ist die Möglichkeit, sehr grosse Datenmengen und sehr grosse Dateien zu verwalten. EXT4 ist abwärtskompatibel zu EXT3 und EXT3 Dateisysteme können in EXT4 umgewandelt werden.

Informationen zu allen Versionen des Dateisystems befinden sich auf http://ext4.wiki.kernel.org.

3.4.9 *Network File System* (*NFS*)

Das *Network File System* (*NFS*) ist hilfreich, um Verzeichnisse auf einem entfernten Rechner, in das lokale Dateisystem einzubinden. Abhängig von den vergebenen Zugriffsrechten kann mit NFS auf das entfernte Verzeichnis zugegriffen werden, als ob es Bestandteil des eigenen Verzeichnisbaumes ist. Da NFS keinen physikalischen Datenträger verwaltet, sondern ein existierendes Dateisystem über ein Netzwerk transportiert und abbildet, handelt es sich um ein sogenanntes virtuelles Dateisystem.

Im Rahmen der Softwareentwicklung für Pervasive Linux ist das besonders hilfreich, um Entwicklungsergebnisse zwischen dem Entwicklungsrechner und der Zielumgebung auszutauschen. Ist beispielsweise ein Verzeichnis der Zielumgebung in den Verzeichnisbaum der Entwicklungsumgebung eingebunden, können die Ergebnisse direkt in diesem Verzeichnis abgelegt werden und sind dann sofort auf der Zielumgebung verfügbar und ausführbar.

Weniger praktikabel ist es, das Verzeichnis des Entwicklungsrechners, in dem die Ergebnisse abgelegt werden, in den Verzeichnisbaum der Zielumgebung einzubinden, da in diesem Fall der Programmcode zum Ausführungszeitpunkt über das Netzwerk geladen wird, was sich abhängig von der Netzwerkgeschwindigkeit negativ auf die Dauer des Startvorgangs auswirken kann.

Das Projekt der Implementierung von NFS für Linux befindet sich unter http://nfs.sourceforge.net/.

3.4.10 *Common Internet File System* (*CIFS*)

Wie NFS ermöglicht das *Common Internet File System* (*CIFS*) als virtuelles Dateisystem den Zugriff auf Verzeichnisse eines entfernten Rechners. Insbesondere wenn Verzeichnisse von Systemen genutzt werden sollen, die nicht aus dem Linux/UNIX-Umfeld stammen, kommt CIFS häufig zum Einsatz. Die Projektseite im Internet ist http://linux-cifs.samba.org/.

3.5 Daten

An jedes datenverarbeitende Gerät wird die Anforderung gestellt, die zu verarbeitenden Daten auf definierte Art und Weise zu lesen und zu schreiben. Dabei muss zunächst die Kodierung bekannt sein, in der die Daten transportiert und möglicherweise gespeichert werden. Der zweite Schritt ist die Definition von Regeln bzw. einer Struktur, wie die Daten organisiert werden.

3.5.1 Zeichenkodierungen

Zeichenkodierungen definieren, wie Schriftzeichen auf numerische Werte innerhalb eines so genannten Zeichensatzes abgebildet werden. Damit ist eine Zuordnung zwischen den in einem Datensatz gespeicherten Zahlen auf die Schriftzeichen möglich, die sie repräsentieren.

3.5.1.1 ASCII

Die Zeichenkodierung *American Standard Code for Information Interchange* (*ASCII*) ist die Basis für die meisten Systeme, um Textdaten zu verarbeiten. Mit 7 Bit wird das lateinische Alphabet sowie einige Sonderzeichen abgebildet. Die Darstellung von Schriftzeichen nicht-lateinischer Sprachen oder Sprachspezifika wie Umlaute oder Akzente sind ohne Erweiterungen nicht möglich.

3.5.1.2 Unicode

Der *Unicode* Standard zur Repräsentation von Textzeichen ermöglicht die Ablage von Texten in pratisch jeder existierenden Sprache mit den entsprechenden Schriftzeichen. Dabei ist für jedes Zeichen ein eindeutiger Code aus zwei Byte (16 Bit) definiert. Unicode-Zeichen werden üblicherweise hexadezimal mit einem vorangestellten U dargestellt, also z. B. `U00C4` für ein Ä.

Das am häufigsten verwendete Kodierungsformat, um Unicode-Zeichen zu persistieren ist *8-bit UCS Transformation Format* (*UTF-8*), wobei es auch eine 16-bit Variante (UTF-16) gibt.

Die Programmiersprache Java (Abschn. 6.2.2) setzt von vornherein auf die interne Repräsentation von Zeichenketten (engl.: String) als Unicode.

Gepflegt wird der Standard vom Unicode Consortium auf der Webseite http://www.unicode.org

3.5.2 Datenformate

Datenformate legen die Struktur von Daten auf einem Datenträger oder zur Übertragung über ein Netzwerk fest, spielen in allen Bereichen der Datenverarbeitung eine Rolle, somit auch im Pervasive Computing.

3.5.2.1 Extensible Markup Language (XML)

Die so genannte Erweiterbare Auszeichnungssprache (*engl.: Extensible Markup Language* (*XML*)) stellt einen Meilenstein im Bereich der Standardisierung von Datenformaten und -organisation dar. XML-Dateien bieten einen praktikablen Mittelweg zwischen Maschinenlesbarkeit und für Menschen lesbarem und verständlichem Inhalt.

Eine XML-Datei enthält eine Hierarchie von Elementen, die den Inhalt der Datei beschreiben. Ein Element wird in spitze Klammern ⟨⟩ gefasst und kann weitere Elemente enthalten. Damit ein XML-Dokument gültig ist, müssen alle Elemente durch ein gleichnamiges Pendant geschlossen werden. Schliessende Elemente haben den selben Namen, beginnen jedoch mit einem vorwärtsgerichteten Schrägstrich.

Beispielsweise könnte ein Name folgendermassen mit XML kodiert werden.

```
<name>
  <vorname>
    Christoph
  </vorname>
  <nachname>
    Czernohous
  </nachname>
</name>
```

Die Einrückung dient in diesem Beispiel dabei ausschliesslich der besseren Lesbarkeit für Menschen. Enthält ein Element keinen expliziten Wert, sondern zeigt durch seine Existenz allein schon einen Wert an, kann auf das schliessende Element verzichtet und der Schrägstrich an das Ende des Elements gesetzt werden. In diesem Fall werden öffnendes und schliessendes Element vereint, z. B. wenn der Wert leer bleibt: $\langle fax - nr/\rangle$.

Zusätzlich zu umschlossenen Elementen, können Elemente auch so genannte Attribute beinhalten. Diese werden als Schlüsselwort-Wert-Paare innerhalb der Elementdefinition angegeben.

```
<rechteck breite=10 hoehe=5/>
```

Welche Elemente und Attribute ein gültiges XML-Dokument enthalten kann oder muss, wird in einer so genannten Schema-Definition (früher Dokument-Typ-Definition) festgelegt. Aus der Schema-Definition ergibt sich das Vokabular für die XML-Dokumente, die auf Basis dieses Schemas erstellt werden, also alle gültigen Elemente samt deren Attribute und möglichen Verschachtelungen. XML-Schema-Dateien haben üblicherweise die Dateiendung `.xsd`.

XML-Dokumente haben eine Reihe an Vorteilen. Durch die lesbaren Elemente können Menschen den Inhalt einer Datei verstehen. Die strenge Forderung nach *Wohlstrukturierung*, also dass jedes öffnende Element durch ein schliessendes beendet wird, ermöglicht sowohl eine prinzipiell gute Verarbeitung durch Maschinen als auch als Folge daraus die leichte Umwandlung in andere Formate.

Ein Nachteil von XML-Dokumenten besteht darin, dass die explizite Lesbarkeit eine erhöhte Dateigrösse zur Folge hat. XML-Dateien sind Textdateien, die vor ihrer Verarbeitung, während oder nach dem Einlesen erst noch auf ihrer Textbasis ausgewertet werden müssen. Diese Art der Textverarbeitung ist relativ rechenintensiv und verbraucht damit viel Ressourcen.

Namensraum

Im Sinne der Modularisierung und Wiederverwendung bestehender Definitionen ist es oftmals sinnvoll, das Vokabular unterschiedlicher Schema-Definitionen in

einem Dokument zu verwenden. Dabei kann es zu Doppeldeutigkeiten kommen, wenn mehrere Schema-Definitionen die gleichen Elemente deklarieren. Um diese Konflikte zu vermeiden, werden in der XML-Datei die Element-Definitionen unterschiedlicher Schema-Dateien über einen eindeutigen *Namensraum* (*engl.: Name Space*) gekennzeichnet. Mit dem Attribut `xmlns` kann in einer Elementdefinition ein Namensraum benannt werden. Dieser Namensraum hat innerhalb des deklarierenden Elements Gültigkeit. Eine Namensraumdefinition besteht aus einer URI, die idealerweise auf die Schema-Definition verweist. Beispielsweise könnte eine Namensraumdefinition `homepage` innerhalb des Elements `springer` folgendermassen aussehen:

```
<springer:name xmlns:homepage='http://www.springer.com/name'>
```

Bei der Elementdeklaration wird der Namensbereich jeweils vor dem Elementnamen angegeben und von diesem mit einem Doppelpunkt getrennt.

3.5.2.2 vCard

Das *vCard* Datenformat spezifiziert einen Standard, um Kontaktinformationen und Adressdaten zwischen Anwendungen auszutauschen. Damit entspricht es einer digitalen Visitenkarte. Das Format wurde Mitte der neunziger Jahre von dem Industriekonsortium *versit* entworfen und als Standard an das *Internet Mail Consortium* (http://www.imc.org) übertragen. Die Spezifikation des Formats findet sich unter http://www.imc.org/pdi/pdiproddev.html bzw. als überarbeiteter Entwurf unter http://datatracker.ietf.org/doc/draft-ietf-vcarddav-vcardrev/. Üblicherweise wird die Dateiendung `.vcf` für vCard-Dateien verwendet.

Die vCard Visitenkarte des Autors sieht folgendermassen aus:

```
BEGIN:VCARD
VERSION:3.0
UID:107
X-QTOPIA-GENDER:1
REV:2010-11-07T17:31:21Z
TEL;TYPE=HOME,VOICE:071590000000
FN:Christoph Czernohous
N:Czernohous;Christoph
LABEL;TYPE=HOME:Silcherplatz 5\, 71106 Magstadt\,
 Germany
ADR;TYPE=HOME:;;Silcherplatz 5;Magstadt;;71106;Germany
TEL;TYPE=HOME:071590000000
TEL;TYPE=HOME;TYPE=CELL:0160-00000000
LABEL;TYPE=WORK:Sch=F6naicher Strasse 220\,
 71032 B=F6blingen\, Germany
ADR;TYPE=WORK:;;Sch=F6naicher Strasse 220;B=F6blingen;;
 71032;Germany
EMAIL;TYPE=INTERNET:c.c@gmx.net
EMAIL;TYPE=INTERNET:cc@de.ibm.com
ORG:IBM Deutschland Research & Development GmbH
END:VCARD
```

Das vCard-Format sieht eine Menge an Eigenschaften vor, die zusammen die Informationen zu einem Kontakt darstellen. Beispielsweise ist `ADR` der Name der Eigenschaft für eine Adresse, auf deren Eintrag weitere Informationen zu der Adresse folgen. Auf den Eigenschaftsnamen folgt, durch Semikolon getrennt, ein optionaler Parameter. Danach kommt, durch einen Doppelpunkt getrennt, mindestens ein zu der Eigenschaft gehörendes Attribut, weitere Attribute werden durch Semikolon getrennt angehängt.

```
Eigenschaftsname[;Parameter]:Wert
```

Um Eigenschaften zu gruppieren, wird ihnen ein gemeinsamer Gruppenname gefolgt von einem Punkt vorangestellt. Auf diese Weise kann der Zusammenhang der Informationen festgelegt werden. Beispielsweise können auf diese Weise geschäftliche und private Kontaktdaten innerhalb eines vCard-Eintrages gebündelt werden:

```
PRIVAT.TEL:07159-0000000
PRIVAT.NOTE:Nur abends zu erreichen.
```

Innerhalb des Formates können auch mehrere vCard-Einträge enthalten sein, die jeweils durch `BEGIN:VCARD` und `END:VCARD` eingerahmt werden, auch das Schachteln von vCard-Einträgen ist möglich. Das vCard Datenformat muss mit einem `BEGIN:VCARD`-Eintrag beginnen.

Beispiele für Parameter von Eigenschaften sind z. B. `TYPE` im obigen Beispiel, aber auch Abweichungen von den Standardeinstellungen bzgl. der Zeichenkodierung ausserhalb des 7-Bit ASCII Raums `(ENCODIG)` oder die für die Eigenschaft gültige Sprache (`LANGUAGE`). Die Spezifikation sieht eine Vielzahl möglicher Eigenschaften vor und definiert die dazugehörenden Parameter.

Binärdaten, wie z. B. Bilder, Videos oder Tondateien können entweder direkt innerhalb eines vCard Eintrags mit der *Base 64* Kodierung enthalten sein oder als externe Datei über eine URL referenziert werden.

3.5.2.3 vCalendar/iCalendar

Für den Austausch von Terminen und Aufgaben hat sich das Format *iCalendar* etabliert. Dieses Format ist eine Weiterentwicklung des *vCalendar*-Formats und ist in RFC 5545 (http://tools.ietf.org/html/rfc5545) definiert. Analog zum vCard-Format ist der Inhalt im iCalendar-Format 7-Bit ASCII Text mit einer Liste von Eigenschaften und der Dateiendung `.ics`. Eigenschaften werden von ihren Werten durch einen Doppelpunkt getrennt.

Um Einträge zu gruppieren, werden sie in `BEGIN/END` Blöcke eingefasst, die wiederum geschachtelt werden können. So muss die erste Zeile in einer iCalendar-Datei `BEGIN:VCALENDAR` und die letzte Zeile `END:VCALENDAR` sein.

Folgendes Beispiel zeigt den Aufbau einer iCalendar-Datei, mit der ein Termin festgelegt wird.

```
BEGIN:VCALENDAR
VERSION:2.0
METHOD:PUBLISH
BEGIN:VTIMEZONE
TZID:Europe/Madrid
BEGIN:STANDARD
DTSTART:19501029T020000
TZOFFSETFROM:+0200
TZOFFSETTO:+0100
RRULE:FREQ=YEARLY;BYMINUTE=0;BYHOUR=2;BYDAY=-1SU;BYMONTH=10
END:STANDARD
BEGIN:DAYLIGHT
DTSTART:19500326T020000
TZOFFSETFROM:+0100
TZOFFSETTO:+0200
RRULE:FREQ=YEARLY;BYMINUTE=0;BYHOUR=2;BYDAY=-1SU;BYMONTH=3
END:DAYLIGHT
END:VTIMEZONE
BEGIN:VEVENT
DTSTART;TZID="Europe/Madrid":20101109T160000
DTEND;TZID="Europe/Madrid":20101109T163000
TRANSP:OPAQUE
DTSTAMP:20101105T113734Z
CLASS:PUBLIC
DESCRIPTION:Besprechung der weiteren Vorgehensweise \
 bzgl. der Pervasive Linux Strategie
SUMMARY:Teambesprechung
LOCATION:Raum A24
UID:68C60D7446CE8996C12577D2003FB31E
END:VEVENT
END:VCALENDAR
```

Termine im vCalendar-Format werden in Blöcke zwischen `BEGIN:VEVENT` und `END:VEVENT` eingefasst. Das gleiche gilt für Aufgabeneinträge, die entsprechend zwischen logischen Klammern `BEGIN:VTODO` und `END:VTODO` stehen müssen.

Für die Einbindung von Binärdaten gelten die selben Regeln wie im vCard-Format. Auch für vCalendar definiert die Spezifikation eine Vielzahl an bekannten Eigenschaften. Zur Definition von sich wiederholenden Terminen steht eine umfangreiche Grammatik bereit.

3.6 Benutzerschnittstellen

Wie bei Linuxsystemen im Desktop-Bereich reicht die Palette an Software, über die der Benutzer mit dem eigentlichen System kommuniziert, von textbasierten Kommandozeileninterpretern (Command-Shells) über einfache grafische Fenstersysteme bis zu integrierten grafischen Oberflächen. Während die gebräuchlichen zeichenorientierten Kommandozeileninterpreter wie Bourne-Shell, C-Shell, Korn-Shell und bash auch für die meisten eingebetteten Linuxumgebungen

verfügbar sind, und es, wenn auch meistens mit reduziertem Funktionsumfang, Implementierungen des X-Window Systems und anderer Fenstersysteme gibt, ist an den Einsatz der beiden verbreitetsten grafischen Oberflächen KDE (http://www.kde.org) und GNOME (http://www.gnome.org) aufgrund deren Ressourcenanforderungen in eingebetteten Linuxsystemen nicht zu denken.

3.6.1 Anmeldung und Benutzerverwaltung mit TinyLogin

Die Aufforderung, sich am System anzumelden, stellt normalerweise den ersten Kontakt zwischen System und Benutzer dar. Nach erfolgreichem Start des Kernels und Initialisierung des Systems sind die konfigurierten Dialogstationen bereit zur Dateneingabe und fordern den Benutzer auf sich anzumelden. Der `getty` Prozess überwacht die konfigurierten Dialogstationen und zeigt die Aufforderung zum Login auf diesen an. Nach erfolgter Eingabe von Benutzer ID und Passwort startet der Terminal-Prozess das Program /bin/login, um die Eingabe zu überprüfen.

Für ressourcenbeschränkte Umgebungen hat sich Implementierung TinyLogin (http://tinylogin.busybox.net) etabliert. Diese Umsetzung des Programms *login* vereint zentrale Funktionen der Benutzerverwaltung wie Authentisierung des Benutzers beim Login-Vorgang, Passwortverwaltung und das Hinzufügen und Entfernen von Benutzern im System. Durch die Realisierung in einer einzigen Binärdatei und Konzentration auf die wichtigsten Funktionen benötigt TinyLogin nur sehr wenig Speicherplatz.

Zum Zeitpunkt der Erstellung dieses Textes stellte die aktuelle Implementierung von TinyLogin folgende Befehle bereit:

adduser	Füge neuen Benutzer zum System hinzu.
addgroup	Füge neue Benutzergruppe zum System hinzu.
deluser	Entferne Benutzer aus dem System.
delgroup	Entferne Benutzergruppe aus dem System.
login	Starte neue Benutzersitzung im System.
su	Wechsle die Benutzer ID oder werde root.
sulogin	Starte neue Benutzersitzung im Ein-Benutzerbetrieb.
getty	Öffne Dialogstation (tty) und starte /bin/login.

3.6.2 BusyBox

BusyBox (http://www.busybox.net) nennt sich selbst „das Schweizer Taschenmesser eingebetteter Linux Systeme“ und stellt eine Sammlung der wichtigsten Linux Befehle in einer einzigen Binärdatei zusammen. Die meisten von BusyBox implementierten Befehle verfügen über einen reduzierten Funktionsumfang im Vergleich zu den Originalen aus dem GNU Projekt (http://www.gnu.org). Das Ergebnis ist ein gelungener Kompromiss aus Funktionsreichtum und Speicherbedarf, der sich sehr gut für Pervasive Linux Systeme eignet. Auf diese Weise lässt sich mit

Hilfe von BusyBox bereits eine verhältnismässig vollständige Linux Umgebung aufbauen.

Der konkret benötigte Befehlsumfang kann individuell konfiguriert werden. Auf diese Weise lässt sich durch erneutes Kompilieren für die Zielplattform eine weitere Reduzierung des Platzbedarfs erreichen.

Die von BusyBox bereitgestellten Befehle werden als Argumente auf der Befehlszeile übergeben:

```
/bin/busybox pwd
```

Mit diesem Befehl wird das aktuell verwendete Verzeichnis des Benutzers ausgegeben. Alternativ kann ein symbolischer Verweis auf BusyBox für jeden Befehl erstellt werden, um den Aufruf zu erleichtern:

```
ln -s /bin/busybox pwd
./pwd
```

3.6.3 XML User Interface Language (XUL)

Auf der Webseite http://www.mozilla.org/projects/xul wird von der Mozilla Organisation das Projekt *XML User Interface Language* (*XUL*) vorangetrieben. Dabei handelt es sich um eine mit XML definierte Auszeichnungssprache zur Beschreibung von Benutzerschnittstellen. Der grosse Vorteil davon ist, dass diese Beschreibungen plattformunabhängig definiert werden. Ausserdem sorgt die explizite Deklaration in einer eigenen Sprache für eine saubere Trennung von Benutzerschnittstelle und Anwendungslogik.

Eine Voraussetzung für den Einsatz von XUL ist die plattformabhängige Laufzeitumgebung, die XUL interpretiert und für die Darstellung sorgt.

Neben der Sprachdefinition stellt das Projekt mit *XULRunner* auch eine Plattform bereit, die in XUL geschriebene Programme ausführt.

3.6.4 X Window System

Das *X Window System* ist das Fenstersystem für Linux/UNIX Systeme mit dem grössten Bekanntheits- und Verbreitungsgrad. Die meisten grafischen Oberflächen unter Linux basieren auf dem X Window System. Es handelt sich um ein einfaches System zur grafischen Darstellung von Fenstern und Bedienelementen. Der Standard wird von der X.Org Foundation unter http://www.x.org gepflegt.

Die Architektur des X Window Systems sieht eine klare Trennung zwischen Darstellung und Anwendungslogik vor. Die Darstellung erfolgt von einem X-Server, zu dem sich Client-Programmer verbinden können. Üblicherweise werden sowohl Anwendungsprogramme als auch der X-Server auf dem selben Rechner ausgeführt, allerdings können sie sich auch auf unterschiedlichen Systemen befinden. In diesem Fall kann die Oberfläche eines Programms auf einem entfernten Rechner angezeigt werden, auf dem ein X-Server läuft.

3.6.5 GIMP Toolkit (GTK+)

Das *GNU Image Manipulation Program* (*GIMP*), zu finden unter http://www.gimp.org, ist ein mächtiges quelloffenes Bildbearbeitungsprogramm. Als Nebenprodukt aus diesem Projekt ging eine umfangreiche auf dem X Window System basierende objektorientierte Programmierbibliothek zur Entwicklung grafischer Benutzeroberflächen hervor, das *GIMP Toolkit* (*GTK+*). Die GTK+ Homepage ist http://www.gtk.org. Die aus dem PC-Umfeld bekannte Arbeitsoberfläche *GNU Network Object Model Environment* (*GNOME*), zu finden unter http://www.gnome.org, basiert auf GTK+.

Das GTK+ Design sieht vor, dass Anwendungen, die GKT+ verwenden, in diversen Programmiersprachen geschrieben sein können.

3.6.6 Enlightenment

Bei *Enlightenment* handelt es sich um eine Sammlung von Bibliotheken zur Anwendungsentwicklung mit der Programmiersprache *C*. Das Projekt ist im Internet unter der Adresse http://www.enlightenment.org zu finden und wird auch mit dem Akronym *e* bzw. *E* referenziert. Aufgrund des geringen Ressourcenverbrauchs eignet sich Enlightenment sehr gut für den Einsatz im Pervasive Computing. Die Basis für die Entwicklung von Software, die auf Enlightenment basiert, sind die *Enlightenment Foundation Libraries* (*EFL*).

Eines der Hauptziele von Enlightenment ist es, die Entwicklung von optisch ansprechender, effizienter und einfach zu benutzender Software zu ermöglichen.

Erklärtes Ziel der EFL es ist, grösstenteils unabhängig voneinander eingesetzt werden zu können. Der Schwerpunkt liegt dabei auf performanter Grafikverarbeitung, allerdings gibt es auch Bibliotheken, die keine grafische Repräsentation benötigen oder anbieten.

Die zentrale Bibliothek von Enlightenment ist die Zeichenoberfläche (engl. Canvas) *Evas*. Eine grundlegende Eigenschaft von Evas ist die Darstellung mittels pixelorientierter Bilddateien (im Gegensatz zu Vektorgrafik). Ausserdem führt Evas über den momentanen Zustand der Oberfläche buch.

Der Layoutmanager von Enlightenment, *Edje*, sieht für den Aufbau der Benutzerschnittstelle bevorzugt eine Anordnung der Bedienelemente mit relativen Koordinaten vor, erlaubt aber auch die absolute Positionierung. Die Definition der Oberfläche erfolgt deklarativ in Dateien mit der Endung `.edc`, die von einem speziellen Übersetzer in ein Binärformat mit der Dateiendung `.edj` gebracht werden. Bewegung und Animation von Elementen wird erreicht, in dem eine Ausgangs- und Zielgrafik angegeben und die für einen flüssigen Übergang benötigten Zwischenschritte von Edje berechnet werden.

Für den nichtgrafischen Infrastruktur-Teil stellen die EFL die Bibliothek *Ecore* bereit, die in sich wiederum in verschiedene Funktionsbereiche aufgeteilt ist. Sie enthält beispielsweise Abstraktionsschichten für dynamisches Laden zusätzlicher Plug-Ins, Interprozesskommunikation oder Netzwerkzugriff.

Neben den beschriebenen zentralen Bibliotheken gibt es noch eine Reihe weiterer Programmbibliotheken für weitere Aspekete der Anwendungsentwicklung mit Enlightenment. Wichtig zu erwähnen ist *Elementary*, die fertige grafische Bedienelemente enthält.

Die Funktionalität von Enlightenment lässt sich modular erweitern.

3.6.7 Qt for Embedded Linux

Mit *Qt for Embedded Linux* entwickelt die Firma Nokia ein Rahmenwerk für die Entwicklung grafischer Anwendungen im Bereich Pervasive Linux. Das Rahmenwerk besteht aus einer Klassenbibliothek für die Sprache C++ und ermöglicht die plattformunabhängige Entwicklung von Anwendungen. Im Internet sind Informationen zu Qt for Embedded Linux unter der URL http://qt.nokia.com/products/platform/qt-for-embedded-linux zu finden.

Für Qt for Embedded Linux gibt es eine Open Source Edition und eine kommerzielle Variante.

3.6.8 Nano-X

Nano-X war früher unter dem Namen *Microwindows* bekannt, wurde jedoch auf Grund von Namenskonflikten mit anderen Produkten umbenannt. Nano-X ist unabhängig von einem konkreten Betriebssystem, da es direkt die Grafikhardware ansteuert, es kann aber auch mit einem Linux Framebuffer-Gerät oder im X Window System betrieben werden.

Für Nano-X stehen zur Zeit zwei Programmierschnittstellen zur Verfügung, die weitgehend zu den Schnittstellen Win32 bzw. X-Windows kompatibel sind, um eine einfache Portierung vorhandener Programme zu ermöglichen.

Nano-X wurde bereits auf die Plattformen IA16, IA32, MIPS, StrongARM und PowerPC portiert.

3.6.9 MiniGUI

MiniGUI wird von der Firma Feynman Software entwickelt und unter der GPL Lizenz vertrieben (http://www.minigui.com). Inhalt ist eine ressourcensparende grafische Benutzeroberfläche für diverse Betriebssysteme, insbesondere eingebettete Echtzeitsysteme. Die Architektur sieht eine weitgehende Abstraktion von Hardware-Plattform und Betriebssystem vor, nimmt aber Rücksicht auf spezielle Restriktionen, wie z. B. der Verfügbarkeit einer Speicherverwaltungseinheit im

Prozessor und damit einhergehenden getrennten Adressbereichen für Prozesse. Aus diesem Grund kann MiniGUI in drei verschiedenen Ausprägungen betrieben werden:

Threads	Dieser Betriebsmodus ist für Plattformen gedacht, auf denen keine Speicherverwaltungseinheit im Prozessor existiert und damit auch ein Prozesskonzept mit getrennten Speicherbereichen im Betriebssystem nicht ohne weiteres umgesetzt werden kann. In diesem Fall werden alle grafischen Speicherstrukturen im selben physischen Adressraum abgelegt.
Prozesse	Unterstützt das Betriebssystem eigenständige Prozesse, sollte dieser Betriebsmodus gewählt werden.
Stand Alone	Falls das verwendete Betriebssystem weder Threads noch Prozesse unterstützt, kann MiniGUI auch in einer eigenständigen Ausführeinheit betrieben werden.

3.6.10 freesmartphone.org (FSO)

Das Projekt *FSO*, zu finden unter http://www.freesmartphone.org, hat den Anspruch, eine offene Diensteplattform auf Linux zu entwickeln, die auf Multifunktionsmobiltelefonen (engl.: Smartphones) einsetzbar ist. Die Architektur von FSO sieht die Unterstützung von Programmen vor, die Enlightenment, Qt, GTK oder X11 als grafische Oberfläche verwenden.

3.6.11 GPE Palmtop Environment (GPE)

Mit dem GPE Palmtop Environment (GPE) steht eine grafische Benutzeroberfläche auf Basis des X-Window Systems und des GIMP Toolkits (GTK) zur Verfügung. Das Paket besteht aus einem Satz Personal Information Management (PIM) Anwendungen, sowie zahlreichen weiteren aus dem Linux-Umfeld bekannten Programmen. Die Homepage von GPE ist http://gpe.linuxtogo.org, Abbildung 3.1 zeigt die Aufteilung der vorkonfigurierten GPE Oberfläche.

Das Ziel von GPE ist es ein Framework bereitzustellen für die Entwicklung und Portierung von Programmen, die für ihre grafische Darstellung das X-Window System benötigen. Da sehr viel existierende Software für Linux und UNIX das X-Window System verwendet, erleichtert das die Portierung nach GPE.

3.6.11.1 Eingabemöglichkeiten

Viele Geräte, die für Pervasive Computing eingesetzt werden, verfügen über keine oder eingeschränkte Tastaturen, mit Hilfe derer der Benutzer Text eingeben und im System navigieren kann. Insbesondere bei PDAs fehlt eine Tastatur häufig ganz und

Abb. 3.1 Die GPE Arbeitsoberfläche

für die Navigation steht eine kleine Anzahl spezieller Funktionstasten bereit. Diese Geräte sind dafür ausgelegt, hauptsächlich über den druckempfindlichen Bildschirm (engl.: Touchscreen) bedient zu werden.

Die Navigation im System erfolgt dann durch gezieltes Anwählen, indem mit einem speziellen Stift, einem Finger o.ä. direkt auf dem Bildschirm an der entsprechenden Stelle leichter Druck ausgeübt wird.

GPE stellt als Tastaturersatz standardmässig zwei Alternativen zur Texteingabe zur Verfügung, eine virtuelle Tastatur sowie ein Konzept zur Handschrifterkennung.

Die virtuelle Tastatur wird am unteren Bildschirmbereich ein- und ausgeblendet und mit dem Eingabstift wie eine normale Tastatur bedient. Für den eher seltenen Fall, dass mehrere Tasten gleichzeitig gedrückt werden müssen, gibt es die spezielle Taste *Mult*, die die synchrone Verwendung mehrerer Tasten simuliert. Abbildung 3.2 zeigt die von GPE angebotene virtuelle Tastatur. Tastaturen für weitere Sprachen können von der GPE Homepage im Internet heruntergeladen und naträglich installiert werden. Es besteht auch die Möglichkeit, eine spezielle Tastatureinteilung selbst zu erstellen.

Für die Handschrifterkennung *Rosetta* wird ebenfalls am unteren Bildschirmrand ein Bereich eingeblendet mit je einem Feld für Kleinbuchstaben, Grossbuchstaben und Ziffern. Abbildung 3.3 verdeutlicht das Prinzip der Handschrifterkennung. Diese Handschrifterkennung ist lernfähig, d.h. der Benutzer kann sie für seine spezielle Handschrift trainieren, bzw. individuelle Strichfolgen bestimmten Zeichen zuordnen.

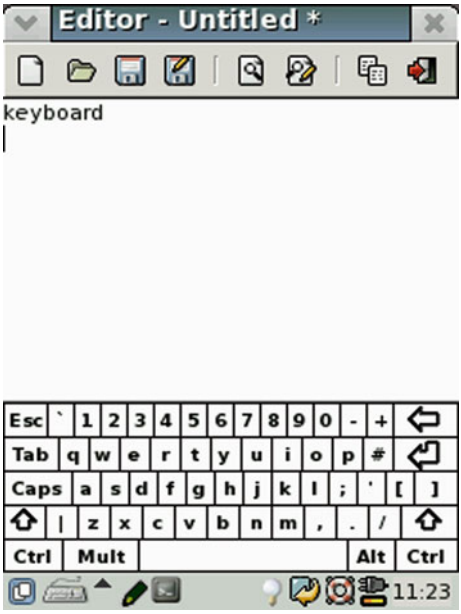

Abb. 3.2 Die virtuelle Tastatur von GPE

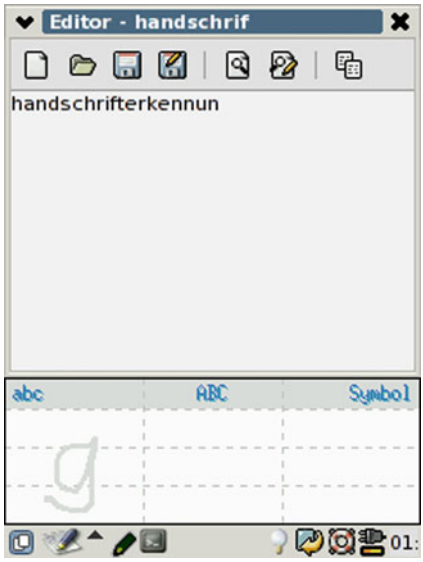

Abb. 3.3 Handschrifterkennung von GPE

3.6.11.2 Oberfläche

GPE stellt Programme, Verzeichnisse und Datendateien auf dem zur Verfügung stehenden Bildschirmbereich mit Hilfe von Symbolbildern (engl.: Icon) dar, wobei am linken oberen Bildschirmrand der Name der Navigationsebene angezeigt wird, in dem sich der Benutzer gerade befindet. Aussehen (Farbwahl, Schriftgröße, etc.) sowie Ausrichtung (horizontal, vertikal) des Anzeigebereichs können vom Benutzer konfiguriert werden.

Abbildung 3.4 zeigt die GPE Oberfläche in horizontaler Ausrichtung.

Falls der sichtbare Bereich der Anzeige nicht ausreicht, um sämtliche in der aktuellen Navigationsebene vorhandenen Elemente anzuzeigen, kann der Benutzer mit den Pfeilsymbolen am rechten oberen Bildschirmrand den sichtbaren Ausschnitt vertikal verschieben.

Die Navigation auf der Oberfläche erfolgt durch Auswählen und Aktivieren des gewünschten Sinnbilds. Dies kann entweder über direktes Aktivieren von Symbolbildern auf der Oberfläche, z. B. durch Berühren der entsprechenden Stelle auf dem druckempfindlichen Bildschirm, oder über die Navigationstasten des verwendeten Gerätes erfolgen, abhängig davon, welche Möglichkeiten das Gerät bereitstellt. Im zweiten Fall wird zuerst der Auswahlfokus auf das gewünschte Sinnbild gelegt und dann durch Betätigen der dafür vorgesehenen Taste aktiviert.

Alternativ zur Navigation direkt auf der Oberfläche kann auch das so genannte Panel zum Starten von Anwendungen verwendet werden. Das Panel wird wie in Abb. 3.5 gezeigt, am unteren oder rechten Bildschirmrand, je nach Bildschirmausrichtung dargestellt. Zum einen bietet das Panel die Möglichkeit, Anwendungen schnell über auf dem Panel platzierte Sinnbilder zu starten, zum anderen kann der Benutzer über aufklappbare Menüs durch die hierarchische Verzeichnisstruktur der Oberfläche navigieren. Die auf dem Panel verfügbaren Sinnbilder zum schnellen Starten häufig verwendeter Anwendungen, können individuell konfiguriert werden.

Abb. 3.4 GPE Arbeitsoberfläche in horizontaler Ausrichtung

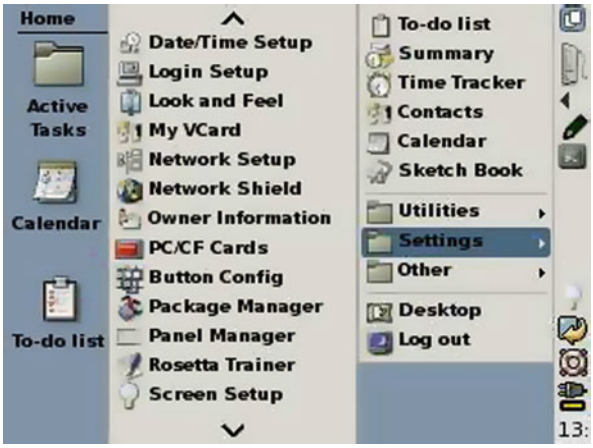

Abb. 3.5 GPE Panel

3.6.11.3 Konfiguration

Die auf der Oberfläche angezeigten Programme und Unterverzeichnisse können vom Systemadministrator (root) konfiguriert werden. Die Regeln für die Konfiguration der Oberfläche basieren auf den vom Projekt Freedesktop (http://www.freedesktop.org) gesammelten Erkenntnissen. Für jedes Unterverzeichnis, das auf der Oberfläche erscheinen soll, muss eine passende Datei mit der Endung `.directory` im Unterverzeichnis `/usr/share/matchbox/vfolders` angelegt werden. Diese Datei spezifiziert den Namen und einen beschreibenden Kommentar des Verzeichnisses, ggf. für unterschiedliche Sprachen, die als Sinnbild zu verwendende Bilddatei sowie die Information, dass es sich bei dem Element um ein logisches Unterverzeichnis handelt. Ausserdem wird ein Schlüsselwort vergeben, dass in entsprechenden Definitionsdateien für Anwendungen verwendet wird, um die zugehörenden Sinnbilder in bestimmten Unterverzeichnissen anzuzeigen. Für das standardmässig konfigurierte Unterverzeichnis `Settings` sieht die Definitionsdatei wie folgt aus:

```
[Desktop Entry]
Name=Settings
Name[de]=Einstellungen
Comment=Settings for your handheld computer
Comment[de]=Einstellungen fuer Ihren Handheld-Computer
Icon=mbfolder.png
Type=Directory
Match=Settings
```

Sinnbilder für Anwendungen werden auf ähnliche Weise wie Unterverzeichnisse konfiguriert. Dazu wird für jede Anwendung im Verzeichnis /usr/share/applications eine Datei mit der Endung .desktop angelegt. Diese Dateien habe einen ähnlichen Aufbau wie die oben beschriebenen Definitionen für logische Unterverzeichnisse.

Auch hier wird ein Name und beschreibender Kommentar vergeben, es wird das zu startende Programm sowie der Bildschirm, auf dem es angezeigt werden soll, spezifiziert. Als Typ des Oberflächenelementes wird diesmal `Application` angegeben, da es sich um eine Anwendung handelt, und die Bilddatei für das Sinnbild wird eingetragen. Unter dem Schlüsselwort `Categories` werden die Unterverzeichnisse aufgelistet, in denen das Symbolbild der Anwendung angezeigt werden soll. Es folgt die Konfigurationsdatei für die installierte Terminal-Anwendung.

```
[Desktop Entry]
Name=Terminal
Comment=Open a new terminal
Exec=x-terminal-emulator
Terminal=0
Type=Application
Icon=gpe-terminal.png
Categories=Utility;GPE
```

3.6.12 Open Palmtop Integrated Environment (OPIE)

Das Projekt *Open Palmtop Integrated Environment* (*OPIE*) hat wie GPE eine grafische Benutzeroberfläche für das Pervasive Computing zum Ziel. OPIE basiert auf dem von der Firma Nokia (Abschn. 3.6.7) entwickelten Fenstersystem Qt sowie der Grafikbibliothek Qt Extended. Diese Software ist inzwischen für Linux unter der GPL als Open Source freigegeben. Abbildung 3.6 zeigt den Aufbau der Benutzeroberfläche von OPIE.

Während GPE die Portierbarkeit vorhandener auf dem X-Window System basierender Programme als Hauptziel hat, konzentriert sich das OPIE Projekt auf einen alltagstauglichen PDA. OPIE bringt daher leistungsfähige Anwendungen u.a. für die Adressverwaltung, Aufgabenliste und einen Kalender für die Terminplanung mit. Die OPIE Oberfläche wurde von der Firma Sharp als Benutzerschnittstelle auf den mit Linux betriebenen PDAs der Reihe „Zaurus“eingesetzt. Die OPIE Plattform kann im Quellcode oder für diverse Zielumgebungen bereits übersetzt von der Seite http://opie.sourceforge.net heruntergeladen werden.

3.6.12.1 Eingabemöglichkeiten

OPIE stellt standardmässig drei Eingabemöglichkeiten bereit, die am unteren Bildschirmrand ein- und ausgeblendet werden können. Wie bei GPE gibt es eine virtuelle Tastatur sowie eine lernfähige Handschrifterkennung. In Abb. 3.7 ist die eingeblendete Tastatur zu sehen.

Zusätzlich zu virtueller Tastatur und Handschrifterkennung bietet OPIE eine weitere Möglichkeit zur Texteingabe, das so genannte Pickboard. Dabei handelt es sich um eine Variante der Tastatur, die weniger Platz auf dem Bildschirm benötigt. Das Pickboard besteht aus zwei Zeilen. Die erste Zeile beinhaltet die wichtigsten Funktionstasten wie Shift, Steuerung, Eingabe oder Entfernen, sowie diverse Sonder-

Abb. 3.6 Die Benutzeroberfläche von OPIE

Abb. 3.7 Die virtuelle Tastatur von OPIE

zeichen. In der zweiten Zeile werden die Buchstaben des Alphabets zu Gruppen von jeweils drei Zeichen zusammengefasst. Ein Kürzel für Ziffern befindet sich ebenfalls in der ersten Zeile. Um nun ein Zeichen einzugeben wird zunächst in der ersten Zeile eine Steuerungstaste gedrückt (z. B. Shift, um einen Grossbuchstaben einzugeben) und dann in der unteren Zeile eine Buchstabengruppe ausgewählt.

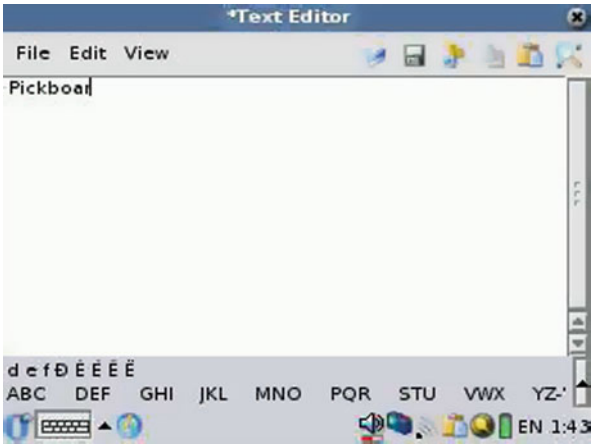

Abb. 3.8 Das OPIE Pickboard

Daraufhin wechselt die Anzeige der oberen Zeile und präsentiert dem Benutzer die hinter der ausgewählten Gruppe zur Verfügung stehenden Zeichen. Das sind normalerweise die drei Buchstaben der gewählten Zeichengruppe, sowie ggf. diverse Umlaute. Abbildung 3.8 zeigt das OPIE Pickboard mit der Auswahl für die Zeichengruppe *DEF*.

3.6.12.2 Oberfläche

Der grösste Teil des zur Verfügung stehenden Bildschirms wird von der Anwendung Launcher eingenommen. Über diese Anwendung können Programme gestartet werden, entweder durch kurzen Druck auf einen druckempfindlichen Bildschirm oder durch Auswahl über ggf. vorhandene Navigationstasten des eingesetzten Gerätes.

Launcher bietet auch die Möglichkeit, Kontextmenüs anzuzeigen, wie sie auf PC Desktop Umgebungen beispielsweise über die rechte Maustaste aktiviert werden. Da auf PDAs und anderen Geräten, auf denen OPIE potenziell eingesetzt werden kann, normalerweise keine Maus vorhanden ist, sieht OPIE die Aktivierung eines Kontextmenüs durch längeren Druck auf dem Bildschirm für ein gewähltes Sinnbild vor. Wird ein Sinnbild nicht nur kurz angetippt sondern bleibt der Druck eine kurze Zeit bestehen, wird nicht die Anwendung gestartet, sondern das Kontextmenü angezeigt.

Die Anwendung Launcher sieht eine flache Navigationsstruktur vor, d.h. Anwendungen werden als Sinnbilder auf einer Ebene dargestellt. Anwendungen sind jedoch in Kategorien gruppiert. Diese Kategorien werden am oberen Bildschirmrand als Auswahlreiter angezeigt. Durch Auswahl eines Reiters werden die Anwendungen dieser Kategorie angezeigt. Können nicht alle Sinnbilder auf dem Bildschirm angezeigt werden, bietet die Anwendung Launcher eine Schiebeleiste am rechten Bildschirmrand an, mit der der sichtbare Bereich verschoben werden kann, wie in Abb. 3.9 zu sehen.

Abb. 3.9 Die Anwendung Launcher

Die letzte Kategorie in der Liste der Auswahlreiter ist für die Ablage von Dateien des Benutzers vorgesehen. In dieser Kategorie können Dateien abgelegt und über die angezeigten Sinnbilder direkt in der zugeordneten Anwendung geöffnet werden.

Am unteren Rand der Anzeige können in der so genannten Taskbar Anwendungen integriert werden, die dem Benutzer sehr komprimiert bestimmte Informationen anzeigen, unabhängig davon, welcher Auswahlreiter im oberen Bildschirmbereich angezeigt wird. Diese Möglichkeit wird z. B. genutzt, um den Benutzer über die aktuelle Uhrzeit, die genutzte Energiequelle oder die Lautstärke der Tonausgabe des Systems zu informieren. Über diese kleinen Anwendungen (Applets) können in der Regel darüber hinaus Konfigurationen vorgenommen werden, indem der Anzeigebereich der gewünschten Anwendung ausgewählt wird. Durch Druck auf die Uhrzeit lässt sich beispielsweise die Systemuhr einstellen.

In der Mitte der Taskbar zeigt OPIE die zum aktuellen Zeitpunkt gestarteten Anwendungen durch Einblenden der zugehörigen Sinnbilder an. Durch Auswahl dieser Sinnbilder kann der Benutzer zwischen den aktiven Anwendungen wechseln, d.h. eine bestimmte Anwendung in den Vordergrund holen, um mit ihr zu arbeiten.

Am linken Rand der Taskbar steht mit dem „O“ eine Navigationsalternative für die Oberfläche bereit. Wählt der Benutzer das „O“-Sinnbild aus, öffnet sich ein Menü, das Zugriff bietet sowohl auf die Auswahlreiter des Launchers und die darin enthaltenen Anwendungen, als auch auf zusätzliche Anwendungen, die nicht auf der Oberfläche vorhanden sind. Abbildung 3.10 zeigt die Funktionsweise des „O“-Menüs.

Inhalte der Taskbar sowie die farbliche und grafische Gestaltung aller Oberflächenelemente können vom Benutzer konfiguriert werden. Zu den bereits vorhandenen Oberflächenstilen besteht die Möglichkeit, eigene Stile zu definieren und zu installieren.

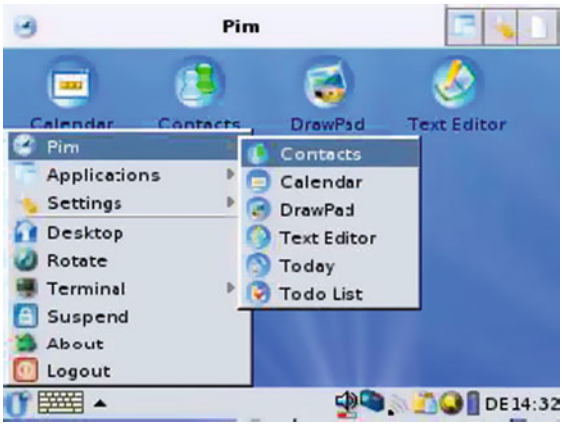

Abb. 3.10 Das O-Menü von OPIE

3.6.13 GNOME Mobile

Mit der GNOME Mobile & Embedded Initiative (GMAE) möchte die GNOME Foundation, die hinter dem erfolgreichen GNOME Projekt steht, die Verwendung von GNOME Komponenten und Technik auf mobile Endgeräte ausweiten.

Die Initiative wurde im Frühjahr 2007 angekündigt und setzt auf bewährte quelloffene Projekte. Informationen über das Projekt sind auf http://www.gnome.org/mobile zu finden.

3.6.14 Clutter

Mit *Clutter* treibt die Firma Intel auf der Web-Seite http://www.clutter-project.org die Entwicklung plattformübergreifenden Grafikbibliothek für das Pervasive Computing voran. Die Implementierung setzt dabei auf den Standard *OpenGL* bzw. *OpenGL ES* (http://www.opengl.org), wobei Clutter das Ziel hat, die Komplexität von OpenGL hinter der eigenen Programmierschnittstelle zu verbergen. U.a. wird Clutter im MeeGo-Projekt eingesetzt.

3.7 Distributionen und Plattformen

Wie im Desktop-Umfeld gibt es inzwischen Zusammenstellungen von Linux-Systemen auch im Pervasive Computing Bereich. Diese beinhalten meistens einen Linux-Kernel sowie zusätzliche für den jeweiligen Einsatzzweck ausgesuchte Anwendungen und zielen auf einen bestimmten Gerätetyp.

3.7.1 OpenWrt

OpenWrt, zu finden auf der Webseite http://openwrt.org/ ist eine Linux-Distribution für eingebettete Systeme. Der Haupteinsatzzweck von OpenWrt ist der Betrieb als Firmware auf einem Netzwerk-Router, das System kann und wird aber auch auf anderen eingebetteten Systemen betrieben.

OpenWrt verfolgt die Philosophie, dass sämtlicher Quellcode auf das Entwicklungssystem heruntergeladen und auf die jeweilige Zielplattform zugeschnitten für den Erstellungsprozess angepasst und inkl. dem benötigten Cross-Compiler übersetzt wird.

3.7.2 Ångström

Das Ziel der Plattform *Ångström* ist eine stabile und benutzerfreundliche Linux-Plattform für eingebettete Systeme. Projekte wie z. B. GPE (Abschn. 3.6.11) basieren auf Ångström. Momentan hat Ångström hauptsächlich PDAs und Tablets als Zielumgebungen.

Ångström-Umgebungen werden mit OpenEmbedded (Abschn. 5.2.14) erstellt.

3.7.3 Puppy Linux

Die Distribution *Puppy Linux* verhilft dem mobilen Einsatz von Linux zur Anwendung insofern, als sich die Installation auf einer CD oder einem USB-Gerät vornehmen lässt. Ein derartiges Installationsmedium wiederum lässt sich sehr leicht transportieren. Natürlich lässt sich Puppy Linux auch wie gewohnt auf einem Gerät installieren.

Ein besonderes Augenmerk legen die Entwickler von Puppy Linux auf eine kleine, effiziente und schnelle Zusammenstellung der Code-Basis und den damit einhergehenden geringen Ressourcenbedarf. Auf Informationen zu Puppy Linux kann auf der Webseite http://puppylinux.org zugegriffen werden. Allerdings fokussiert sich Puppy Linux auf den Einsatz auf Personal Computern, nimmt also insofern eine Sonderrolle im Bereich Pervasive Linux ein.

Egal, ob Puppy Linux direkt von CD aus gestartet wird, oder auf ein anderes Medium installiert wird, muss zuerst ein so genanntes ISO-Image, also das auf die CD zu brennende Abbild der Distribution heruntergeladen und auf die CD gebrannt werden. Nachdem dem Start von Puppy Linux von CD, werden hardware- und benutzerspezifische Einstellungen abgefragt. Beim Herunterfahren des Systems hat man die Möglichkeit, diese Einstellungen in einer Datei mit dem Namen `pup_save.2fs`, zum Beispiel auf einer Festplatte oder einem USB-Gerät zu speichern. Wenn die Boot-CD mehrere Schreibvorgänge zulässt können die Einstellungen auch direkt auf der CD abgelegt werden.

Puppy Linux wird nach dem Start vollständig aus dem Hauptspeicher des PCs ausgeführt, es sei denn, der PC hat weniger Hauptspeicher als 256 Megabyte.

3.7.4 Android

Android ist ein in mehrere Schichten unterteiltes Softwarepaket für mobile Endgeräte. Initiiert und massgeblich vorangetrieben wird es von der Firma Google auf der Seite http://www.android.com/. Es beinhaltet das Betriebssystem, eine so genannte Middleware und eine Reihe an Anwendungen für den Endbenutzer. Auf der Webseite stehen auch die für die Entwicklung benötigten Bibliotheken und Werkzeuge (*engl.: Software Development Kit* (*SDK*)) inkl. einem Plug-In für Eclipse (Abschn. 5.4), den so genannten *Android Development Tools* (*ADT*), bereit. In einem speziellen Emulator können Android-Abbilder (*Android Virtual Device* (*AVD*)) auf Basis der ARM-Plattform betrieben werden.

Android basiert auf einem Linux-Kernel und bietet eine Reihe an Programmbibliotheken um Basisfunktionalität wie Grafik, Netzwerk oder beispielsweise Schriftarten zu verwenden.

Weiterer Kernbestandteil von Android ist die Implementierung einer virtuellen Maschine für die Java-Plattform (Abschn. 6.2.2) namens *Dalvik*. Auf diesen Komponenten baut die Anwendungsschicht auf, die Funktionalität für die Anwendungsentwicklung bietet.

Dieses Anwendungsrahmenwerk wird von Anwendungen genutzt, von denen einige bereits Bestandteil von Android sind. Daher kommt für die Anwendungsentwicklung hauptsächlich die Programmiersprache Java zum Einsatz.

Zur Zeit sind nicht alle Bestandteile von Android als Open Source veröffentlicht.

3.7.5 MeeGo

MeeGo ist eine Linux-Plattform für mobile Geräte zu finden unter http://meego.com. Das Projekt wird von der *Linux Foundation* (zu finden unter http://www.linuxfoundation.org) betreut und ist aus den Initiativen *Moblin* der Firma Intel und *Maemo* der Firma Nokia hervorgegangen. Beide Projekte hatten das Ziel, eine offene Plattform auf Linux-Basis für Netbooks, MIDs und andere mobile Geräte bereitzustellen. Dieses Ziel wird nun mit MeeGo weiterverfolgt.

3.7.6 Qt Extended

Auf Basis von Qt for Embedded Linux erstellte die Firma Nokia eine Anwendungsplattform für Geräte des Pervasive Computing mit Linux als Betriebssystem. *Qt Extended*, hervorgegangen aus dem Produkt *Qtopia*, beinhaltet eine eigene Implementierung zur Fensterverwaltung (engl.: Window Manager), so dass ein X-Server nicht benötigt wird.

Version 4.4.3 von Qt Extended ist die letzte Version von Qt Extended als eigenständige Plattform. Die Kernfunktionalität von Qt Extended soll in die Qt Basis integriert werden. Ausserdem gibt es ein Projekt namens *QT Mobility* (http://labs.trolltech.com/page/Projects/QtMobility) mit dem die QT Bibliothek um Funktionalität für mobile Geräte erweitert werden soll.

3.7.7 Linaro

In der gemeinnützigen Organisation *Linaro* haben sich Firmen aus der IT-Branche zusammengeschlossen, die im Pervasive Computing tätig sind, um der Zersplitterung des Angebots von Open Source in diesem Umfeld entgegenzuwirken und eine gemeinsame Basis für die Entwicklung zu anzubieten. Auf der Seite http://www.linaro.org werden die Entwicklungsaktivitäten koordiniert. Dabei wird gezielt in existierende quelloffene Projekte investiert, indem Entwicklungsergebnisse an die Projekte zurückfliessen. Auf diese Weise gehen Verbesserungen über den regulären Entwicklungsweg existierende Distributionen ein.

Der Schwerpunkt der Tätigkeit liegt auf Verbesserungen der Linux-Plattform (Kernel, Grafiksubsystem, Multimedia, Energieverwaltung) und der Toolchain. Als Ziel-Hardware unterstützt Linaro momentan die ARM-Plattform.

3.8 Datenaustausch

Die Idee des Pervasive Computing sieht geradezu eine Explosion der Anzahl mobiler und vernetzter Geräte vor. In den allermeisten Fällen sollen zwischen diesen Geräten Daten ausgetauscht und auf dem aktuellen Stand gehalten werden. Beispielsweise ist es erstrebenswert, den persönlichen Adressdatenbestand allen eingesetzten Geräten und Anwendungen wie Mobiltelefon, PDA und E-Mail Programmen verfügbar zu machen und aktuell zu halten. Ausserdem soll auch mit Fremdgeräten kommuniziert werden können und Datenaustausch stattfinden, beispielsweise um eine Telefonnummer oder eine Adresse weiterzugeben.

Diese Tatsache stellt eine grosse Herausforderung für die Verwaltung, Verteilung, Aktualisierung und Synchronisation der verwendeten Daten dar. Im Idealfall soll es möglich sein, eine Datenänderung auf einem beliebigen Gerät vorzunehmen und diese mit minimiertem zeitlichen Verzug und Konflikt beim Abgleich auf die übrigen Geräte zu verteilen, um die Datenkonsistenz zu gewährleisten. Je grösser die Anzahl beteiligter Endpunkte ist, desto schwieriger ist dieses Ziel aufgrund der steigenden Komplexität zu erreichen.

3.8.1 SyncML

Ursprünglich von der *SyncML Initiative* entworfen, wird dieser XML-Standard für den Datenabgleich inzwischen von zwei Arbeitsgruppen der *Open Mobile Alliance* (http://www.openmobilealliance.org) weiterentwickelt und gepflegt. Die beiden Gruppen beschäftigen sich mit den Themen *Datensynchronisation* und *Geräteverwaltung* (engl.: Data Synchronization and Device Management).

SyncML wird von zahlreichen mobilen Geräten und Software für die Verwaltung von persönlichen Informationen unterstützt.

3.8.2 Funambol

Funambol wird von der gleichnamigen Firma entwickelt und über die Webseite https://www.forge.funambol.org/ zur Verfügung gestellt. Es handelt sich hierbei um eine in der Programmiersprache Java geschriebene Synchronisationsplattform, für die es sowohl eine Open Source Lizenz, als auch eine kommerzielle Variante, gibt.

Als Synchronisationsprotokoll setzt die Funambol-Implementierung SyncML ein, der Datenabgleich ist also zwischen beliebigen SynchML Endpunkten möglich.

Die Architektur von Funambol sieht diverse Erweiterungsmöglichkeiten der Plattform mittels so genannter *Module* vor, die in die Umgebung eingebunden werden:

SyncSource:	Geräte und Software, die nicht von sich aus bereits SyncML unterstützen, können über Syncsource-Module in die Plattform eingebunden werden. Eine SyncSouce repräsentiert eine spezifische Datenquelle für die Synchronisation, beispielsweise ein bestimmtes Endgerät oder eine Anwendungssoftware.
Synclets:	Mittels so genannter Synclets können die eingehenden Synchronisationsdaten vor und nach der eigentlichen Synchronisation bearbeitet werden.
Officer:	Über Officer-Erweiterungen werden Authorisierungs- und Authentifizierungssysteme eingebunden.
Admin Web Service:	Zugriff auf die Administrationsschnittstelle des Servers wird über einen Web Service erreicht. Auf diese Weise lässt sich die Administrationsfunktionalität auf einfache Weise in eine bereits vorhandene Administrationsoberfläche integrieren.

Durch diese offene, flexible und erweiterbare Architektur ist es möglich, Datensynchronisation zwischen fast beliebigen Geräten zu ermöglichen.

3.8.3 OpenOBEX

Der von der Infrared Data Association entwickelte Standard *OBEX* (engl.: Object Exchange) dient der Übertragung von binären Daten und wird auch als IrOBEX (Abschn. 4.5.3.5) bezeichnet. Die Spezifikation kann von der IrDA angefordert werden. Der ursprüngliche Einsatzzweck war der Datenaustausch über Infrarot-Verbindungen, das Protokoll wird inzwischen aber auch für Übertragungen mittels Bluetooth (Abschn. 4.6) und SyncML (Abschn. 3.8.1) verwendet.

Auf der Seite http://dev.zuckschwerdt.org/openobex/ gibt es mit *OpenOBEX* eine quelloffene Implementierung des OBEX Standards.

3.8.4 OpenSync

Die Open Source Software *OpenSync* stellt auf der Webseite http://www.opensync.org/ ein Rahmenwerk für den Datenabgleich zwischen zwei Datenquellen bereit.

Datenquellen können physikalische Geräte oder logische Einheiten wie ein Dateiverzeichnis sein. Für die Datenübertragung wird der Standard *OBEX* (Abschn. 4.5.3.5) benutzt.

Zur Einbindung von Synchronisationsquellen in die Plattform verwendet OpenSync einen so genannten *Plug-In*-Mechanismus. Dabei kapselt ein Plug-In eine Datenquelle. Über Plug-Ins kann OpenSync auch um weitere Dateiformate und Verbindungsarten erweitert werden.

Um OpenSync einsetzen zu können, benötigt man die OpenSync-Bibliothek, mindestestens ein Plug-In für die zu synchronisierende Datenquelle und ein Programm, das eine Benutzerschnittstelle bereitstellt.

3.8.5 Synthesis

Das Synthesis-Projekt (http://www.synthesis.ch/), entwickelt von der gleichnamigen Firma, stellt eine Bibliothek zur Verfügung, mit der Anwendungen Daten mittels des SyncML-Standards abgleichen können.

3.8.6 SyncEvolution

SyncEvolution ist eine von Patrick Ohly initiierte Synchronisationssoftware auf Basis des Standards SyncML (Abschn. 3.8.1). Mit SyncEvolution können PIM-Daten zwischen SyncML-fähigen Partnern abgeglichen werden. Das Projekt ist unter http://syncevolution.org/ zu finden. SyncEvolution kann Daten entweder über das Protokoll HTTP mit einem SyncML-Server oder seit der Version 1.0 direkt über Bluetooth mit einem entsprechend SyncML-fähigen Gerät synchronisieren. Eine Datenquelle repräsentiert ein Adressbuch, einen Kalender, eine Aufgabenliste oder eine Merkzettelliste.

SyncEvolution bietet eine Benutzerschnittstelle auf der Kommandozeile an, der Befehl lautet `syncevolution`. Wird der Befehl ohne Parameter aufgerufen, zeigt syncevolution alle verfügbaren Synchronisationsquellen und Hilfetexte zu möglichen Parametern an.

SyncEvolution arbeitet mit so genannten Konfigurationen. Für jede zu synchronisierende Quelle muss eine Konfiguration existieren. Dabei wird eine Konfiguration immer innerhalb eines bestimmten Kontextes ausgeführt. Die Regeln innerhalb eines Kontextes gelten für alle beteiligten Synchronisationspartner. Mit Hilfe des Kontextes wird festgelegt, welche Informationen synchronisiert werden sollen. Alle an der Synchronisation beteiligten Partner synchronisieren die selben Informationen, beispielsweise die Kalenderdaten. Wird kein spezielle Kontext angegeben, gilt der Standardwert `@default`.

Teil II
Softwareentwicklung

Kapitel 4
Netzwerk

Eine der wichtigsten Eigenschaften von Computersystemen heutzutage ist ihre Fähigkeit, Daten mit anderen Geräten auszutauschen. Das gilt ganz besonders für das Pervasive Computing, da die jeweiligen Geräte ihre Aufgabe oft in Zusammenarbeit mit anderen erfüllen.

In diesem Kapitel werden Übertragungsstandards und -protokolle vorgestellt, die sich im mobilen Einsatz bzw. im Pervasive Computing etabliert haben.

4.1 Ethernet

Als Standard zum Aufbau eines lokalen Netzwerks (*engl.: Local Area Network – LAN*) hat sich das System mit dem Namen *Ethernet* durchgesetzt. Linux bietet eine breite Unterstützung der verfügbaren Hardware. Wenn das Zielgerät über eine Ethernet-Schnittstelle verfügt, die von Linux unterstützt wird, ist zumindest während der Entwicklung die Kommunikation auf diesem Weg die erste Wahl.

In einem Ethernet-Netzwerk teilen sich alle beteiligten Geräte das Netzwerkmedium gleichzeitig. Dadurch kann es zu Kollisionen bei der Datenübertragung kommen, wenn mehrere Geräte gleichzeitig senden. Der Standard regelt diesen Fall, indem die Kollision erkannt werden muss und der Sendevorgang nach einer zufällig bestimmten Zeit wiederholt wird. Die Problematik von Kollisionen wird inzwischen auch mit Hilfe von weiterentwickelter Hardware, wie z. B. so genannte *Switches* auf andere Art gelöst und ist daher heute kein praxisrelevanter Nachteil mehr.

Bei der Übertragungsgeschwindigkeit können, entsprechende Hardware vorausgesetzt, bis zu 10 GBit/s erzielt werden. Die Daten werden dabei seriell übertragen.

Gerätedateien für Ethernet-Schnittstellen werden mit *eth* gekennzeichnet, also beispielsweise `/dev/eth0` für die erste Ethernet-Schnittstelle im System. Die Konfiguration von Netzwerkschnittstellen erfolgt mit dem Werkzeug `ifconfig`. Wird `ifconfig` ohne Parameter aufgerufen, zeigt es den aktuellen Zustand aller im System aktiven Netzwerkschnittstellen an.

```
ifconfig
eth0      Link encap:Ethernet  Hardware Adresse 00:1e:37:8c:3a:76
          inet Adresse:192.168.24.101  Bcast:192.168.24.255  Mask
```

C. Czernohous, *Pervasive Linux*, X.systems.press,
DOI 10.1007/978-3-540-68426-8_4, © Springer-Verlag Berlin Heidelberg 2012

```
e:255.255.255.0
          inet6-Adresse: fe80::21e:37ff:fe8c:3a76/64  Gueltigkeit
sbereich:Verbindung
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metrik:1
          RX packets:135498 errors:0 dropped:0 overruns:0 frame:0
          TX packets:70595 errors:0 dropped:0 overruns:0 carrier:
0
          Kollisionen:0 Sendewarteschlangenlaenge:1000
          RX bytes:135728942 (135.7 MB) TX bytes:9456755 (9.4 MB)
          Interrupt:20 Speicher:fe200000-fe220000

lo        Link encap:Lokale Schleife
          inet Adresse:127.0.0.1  Maske:255.0.0.0
          inet6-Adresse: ::1/128  Gueltigkeitsbereich:Maschine
          UP LOOPBACK RUNNING  MTU:16436  Metrik:1
          RX packets:140 errors:0 dropped:0 overruns:0 frame:0
          TX packets:140 errors:0 dropped:0 overruns:0 carrier:0
          Kollisionen:0 Sendewarteschlangenlaenge:0
          RX bytes:10800 (10.8 KB) TX bytes:10800 (10.8 KB)
```

Mit `ifconfig` können Netzwerkschnittstellen aktiviert und deaktiviert werden. Ausserdem erfolgt die Konfiguration des verwendeten Netzwerkprotokolls, meistens TCP/IP, über `ifconfig`.

4.2 Serielle Schnittstelle

Über die *serielle Schnittstelle* besteht meistens die erste Möglichkeit, sich mit dem Zielsystem zu verbinden. Genau genommen befinden sich in den meisten Systemen diverse Schnittstellen, über die eine serielle Kommunikation stattfindet. Serielle Kommunikation bedeutet, im Gegensatz zur *parallelen* Kommunikation, dass der Inhalt eines binären Datenstroms Bit für Bit hintereinander und nicht mehrere Bits gleichzeitig übertragen wird. Mit dem Begriff der *seriellen Schnittstelle* wird im Normalfall ein Anschluss bezeichnet, der den Standard *RS-232* (Recommended Standard 232) unterstützt, welcher die physikalischen Eigenschaften für derartige Verbindungen festlegt.

Unter Linux beginnen die Namen von Gerätedateien des Types RS-232 üblicherweise mit *tty*, beispielsweise `/dev/ttyS0`, eine Abkürzung eines der frühen Anwendungsbereiche, der so genannten Teletyper.

Die zu übertragenden Daten werden in logische Einheiten, so genannte Zeichen (engl.: *Character*) zerlegt. Ein Zeichen besteht aus einer bestimmten Anzahl an Bits. Die Übertragung erfolgt asynchron, d.h. dass auf beiden Seiten die gleiche Taktung für die Dauer der Übertragung eines Zeichens verwendet werden muss und nach der Übertragung keine Rückmeldung an den Sender erfolgt. Es wird also eine feste Zeitdauer definiert, die für die Übertragung jedes Bits verwendet werden soll und innerhalb derer der Sender ein Bit an den Empfänger übertragen kann.

Mittlerweile wird die RS-232-Schnittstelle allerdings immer mehr durch den Standard *Universal Serial Bus* (*USB*) verdrängt.

4.2.1 Konfiguration

Für eine Verbindung über eine serielle Schnittstelle müssen diverse Kommunikationsparameter bekannt sein, auf die sich die an der Verbindung beteiligten Geräte einigen müssen.

Datenbits Die Anzahl an (Nutz-) Datenbits innerhalb eines Zeichens kann zwischen 5 und 9 variieren. Meistens werden 8 Bits verwendet.

Parität Das Konzept der *Parität* dient der einfachen Erkennung von Übertragungsfehlern. Dabei wird den Datenbits ein zusätzliches Paritätsbit angehängt, dessen Wert so gesetzt wird, dass die Anzahl an Bits mit dem Wert 1 einschliesslich des Paritätbits in jedem Zeichen eine gerade oder ungerade Zahl ergibt. Mögliche Konfigurationswerte für die Parität sind:

- N Keine (engl.: None) Parität wird verwendet, es gibt kein Paritätsbit.
- O Die Anzahl an Bits mit dem Binärwert 1 ist immer ungerade (engl.: Odd).
- E Die Anzahl an Bits mit dem Binärwert 1 ist immer gerade (engl.: Even).
- M Das Paritätsbit ist vorhanden, wird aber immer gesetzt (engl.: Mark).
- S Das Paritätsbit wird immer leer (engl.: Space) gelassen.

Stoppbits Bei asynchroner Datenübertragung dienen Stoppbits dazu, Sender und Empfänger zu synchronisieren, da die vereinbarte Taktung auf beiden Seiten nie exakt gleich ist, und somit die Sende- und Empfangszeiten auseinanderlaufen.

Geschwindigkeit Die Geschwindigkeit definiert die Anzahl an Bits pro Sekunde für die Übertragung und hängt von der eingesetzten Hardware ab. Es handelt sich um den Bruttowert, also einschliesslich Paritäts- und Stoppbits, die Nettoübertragungsrate ist also immer geringer.

Datenflusssteuerung Für den Fall, dass das empfangende Gerät die Daten nicht so schnell verarbeiten kann, wie neue Daten gesendet werden, gibt es verschiedene Möglichkeiten, den Datenfluss zu steuern, d.h. so lange zu unterbrechen, bis weitere Daten verarbeitet werden können:

RTS/CTS Bei der Methode *RTS/CTS* signalisiert der sendende Teil seine Bereitschaft bzw. seinen Wunsch, Daten zu schicken (engl: Request To Send – RTS), indem er die dafür vorgesehene Leitung aktiviert. Wenn der Empfänger bereit für den Datenempfang ist, aktiviert er die entsprechende Leitung (engl.: Clear To Send – CTS). Erst dann beginnt der Sender mit der Übertragung.

XON/XOFF Ohne zusätzliche Leitungen kommt die Methode *XON/XOFF* aus. Dabei werden die Sonderzeichen *XON* und *XOFF* – sinngemäss für „Übertragung ein" und „Übertragung aus" – vom Empfänger zum Sender, also in umgekehrter Richtung geschickt. Sobald der Sender das Steuerzeichen *XON* empfängt, beginnt er mit der Übertragung von Daten, bis der Empfänger ein *XOFF* schickt.

4.3 Universal Serial Bus (USB)

Mit der Spezifikation einer universellen seriellen Busarchitektur (engl.: *Universal Serial Bus – USB*), wurde ein Industriestandard für die serielle Kommunikation zwischen unterschiedlichen Geräten geschaffen. Hauptziele der Spezifikation waren die Verbindung zwischen PC und Telefon, die Vereinheitlichung von Verbindungsmöglichkeiten von Personalcomputern, die Bereitstellung einer einfachen Möglichkeit zur Erweiterung und Konfiguration von Systemen und die Vervielfachung der Anzahl verfügbarer Anschlüsse für Erweiterungen.

Mit der Version 2.0 des USB Standards kamen noch die gestiegenen Anforderungen an die Übertragungsrate des Datenaustauschs zwischen zwei Systemen dazu, da sich zum einen die üblicherweise ausgetauschte Datenmenge aufgrund der gestiegenen Leistungsfähigkeit der Systeme deutlich vergrössert hatte und zum anderen neue Gerätetypen wie z. B. externe Festplatten oder USB-Sticks aufkamen, die einen schnellen Datenzugriff erforderten. Inzwischen sind Datenübertragungsraten bis zu 480 Megabit pro Sekunde möglich.

Mittlerweile verfügen die meisten neuen Geräte über eine USB-Schnittstelle, und der Standard verdrängt immer mehr die herkömmliche serielle Schnittstelle (Abschn. 4.2). Die USB-Spezifikation wird von der Organisation *USB Implementers Forum* auf der Webseite http://www.usb.org veröffentlicht.

Die Architektur von USB definiert einen Bus, an den bis zu 127 Geräte angeschlossen werden können. Der Bus wird von einem zentralen Gerät, dem so genannten *Host Controller* gesteuert. USB Geräte sind entweder so genannte *Hubs* oder funktionale Geräte. Mit Hilfe von USB Hubs lässt sich die Anzahl verfügbarer physikalischer USB Anschlüsse vergrössern, wobei jeder Hub als eigenes Gerät behandelt, und somit die Anzahl logischer Geräte, die an den Bus angeschlossen werden können, um eins verringert wird. Bis zu fünf Hubs können in Reihe

hintereinander angeschlossen werden, so dass mit dem Host Controller und den am letzten Hub in der Reihe angeschlossenen funktionalen Gerät eine Kette von maximal sieben Geräten entsteht. Beim Anschluss eines Gerätes an den Bus, wird diesem vom Host Controller eine eindeutige Identifikationsnummer zugewiesen, über die das Gerät im Bus angesteuert werden kann.

Jedes USB Gerät enthält einen Gerätedeskriptor (engl.: Device Descriptor) mit allgemeinen Informationen zu dem Gerät. Beim Anschluss des Gerätes an den Bus liest der Host Controller den Gerätedeskriptor aus und hält die Informationen für das Betriebssystem bereit.

Eine weitere Eigenschaft des USB-Standards ist die Unterstützung des so genannten *Hot Plugging*. Damit ist gemeint, dass zusätzliche Geräte an ein System angeschlossen werden, um dessen Funktionalität zu erweitern, ohne das gesamte System neu starten zu müssen. Idealerweise funktioniert Hot Plugging ohne die Installation zusätzlicher Software wie z. B. Gerätetreiber, so dass der Benutzer nur minimal oder gar nicht tätig werden muss. Das gilt auch für die Stromversorgung der angeschlossenen Geräte. In vielen Fällen erfolgt die Stromversorgung angeschlossener Geräte ebenfalls über den USB Anschluss, wobei einem Gerät bis zu 500 mA zur Verfügung stehen. Das ist gerade für mobile Geräte interessant, da in diesem Fall sowohl ein externes Netzteil, als auch eine zusätzliche Anschlussmöglichkeit für die Stromversorgung entfallen kann und somit kostbares Volumen und Gewicht eingespart werden.

Im Zusammenhang mit Pervasive Linux gibt es mehrere unterschiedliche Aspekte für den Einsatz von USB:

- Die Zielumgebung stellt über USB eine Netzwerkverbindung mit dem Internetprotkoll TCP/IP zu einem anderen Rechner her.
- Das Zielsystem wird mit einer über USB aufgebauten Terminalverbindung angesprochen.
- Der nichtflüchtige Speicher des Zielsystems wird als externes Speichermedium an die Hostumgebung angebunden.
- USB Geräte werden als Erweiterungen an das Zielsystem angeschlossen.
- Die Stromversorgung des Zielgerätes erfolgt über USB.

Ein wichtiger Bestandteil der USB-Spezifikation gerade für das Pervasive Computing ist *USB On-The-Go* (*OTG*), also die USB-Unterstützung für mobile Geräte, die miteinander kommunizieren wollen, ohne dass ein expliziter Host existiert.

4.3.1 Linux USB

Das Projekt zur Entwicklung des USB Subsystems für Linux wird auf der Webseite http://www.linux-usb.org organisiert. Inzwischen wurde die Implementierung in die offiziellen Quellen des Linux-Kernels integriert und dort weiter vorangetrieben.

4.3.1.1 usbfs

Der Dateisystemtyp *usbfs* erlaubt den Zugriff auf USB-Geräte unter Linux. Es handelt sich dabei um ein virtuelles Dateisystem, das üblicherweise an der Stelle `/proc/bus/usb` in das Dateisystem eingehängt wird.

4.3.1.2 usbnet

Für Netzwerkverbindungen über USB unter Linux steht der *usbnet* Gerätetreiber zur Verfügung. Er ist Bestandteil der Linux-Kernelquellen und muss entsprechend als Modul oder fester Kernelbestandteil übersetzt werden.

4.3.1.3 USB Gadget API Framework

Das *USB Gadget API Framework* adressiert den Einsatz von Linux auf der Seite der Geräte, die an einen USB-Host angeschlossen werden.

4.4 Wireless LAN (WLAN)

Ein drahtloses lokales Netzwerk (*engl.: Wireless Local Area Network – WLAN*) ist die gängigste Möglichkeit, über Funk eine Netzwerkverbindung mit hohen Übertragungsraten aufzubauen. Die Funktionsweise soll der von kabelgebundenen Netzwerken wie Ethernet möglichst ähnlich sein.

Damit eine WLAN Verbindung zustande kommt, nimmt ein Gerät über Funk Kontakt zu einem anderen WLAN-fähigen Gerät auf. Normalerweise kontaktiert ein Endgerät einen Router, es können aber auch zwei Endgeräte direkt miteinander kommunizieren. Die Namen der Gerätedateien unter Linux beginnen in den meisten Fällen mit *wlan* oder *wifi*, also z. B. `/dev/wlan0`.

Jedes WLAN wird über eine Kennung identifiziert, der so genannten *Service Set Identifier* (*SSID*). Prinzipiell gibt es für WLANs zwei Betriebsarten, *Managed* und *Ad-Hoc*. Die Betriebsart Managed sieht eine zentrale drahtlose Station vor, mit der sich die Netzwerkteilnehmer verbinden (engl.: Access Point), um in das drahtlose Netz integriert zu werden. Bei einem Ad-Hoc Netzwerk schliessen sich zwei oder mehr Geräte zu einem WLAN ohne zentrale Steuereinheit zusammen. In der Betriebsart Managed kann ein teilnehmendes Geräte unterschiedliche Aufgaben übernehmen.

Master	Das Gerät fungiert als zentrale Steuereinheit im Netzwerk.
Repeater	Die empfangenen Datenpakete werden zwischen anderen beteiligten Geräten weitergereicht.
Secondary	Falls das Master-Gerät ausfällt, übernimmt das Secondary-Gerät die Aufgaben des Master-Gerätes.
Monitor	Das Gerät ist nicht im drahtlosen Netzwerk eingegliedert, sondern überwacht alle Datenpakete, die auf einer bestimmten Frequenz gesendet werden.

Die meisten drahtlosen Netze verwenden den Standard *IEEE 802.11*, der von der usprünglichen Spezifikation *IEEE 802.11* mit eine Übertragungsrate von maximal 2 MBit/s über *IEEE 802.11a*, *IEEE 802.11b*, *IEEE 802.11g*, *IEEE 802.11h* bis *IEEE 802.11n* mit einer Übertragungsgeschwindigkeit von maximal 600 MBit/s weiterentwickelt wurde.

4.4.1 Sicherheit

Da die Funkverbindung zwischen den beteiligten Geräten in einem WLAN räumlich nur durch die Signalstärke begrenzt ist, sollten Sicherheitsvorkehrungen getroffen werden, um ein Ausspähen der Datenverbindung zu verhindern. Der wichtigste Schritt dazu ist die Verschlüsselung der Verbindung. Dazu stehen eine Reihe an Verschlüsselungsalgorithmen mit unterschiedlicher Verschlüsselungsstärke zur Verfügung, die im Laufe der Zeit entwickelt wurden. Dabei wird der Datenverkehr über den Verschlüsselungsalgorithmus und einem beiden Partnern bekannten Schlüssel ver- und entschlüsselt.

WEP *Wired Equivalent Privacy* ist die einfachste, aber auch schwächste Verschlüsselungstechnik. WEP bietet einige Schwachstellen, die Angriffspunkte für potenzielle Einbrecher bieten. Beispielsweise kann bei der Schlüssellänge zwischen 64 und 128 Bit gewählt werden. Insbesondere Schlüssel mit einer Länge von 64 Bit lassen sich sehr schnell errechnen.

WPA Bei *Wifi Protection Access* wird der vereinbarte Schlüssel nur initial benötigt. Anschliessend wird der verwendete Schlüssel automatisch für jedes Datenpaket festgelegt. Dadurch wird ein Eindringen in das WLAN fast unmöglich gemacht.

WPA2 Bei WPA2, der Weiterentwicklung von WPA, wird u.a. der Verschlüsselungsalgorithmus Advanced Encryption Standard (AES) eingesetzt sowie weitere Verschlüsselungsprotokolle implementiert.

Eine weitere Möglichkeit zur Erhöhung der Sicherheit ist das Verbergen der SSID. Dadurch ist der Netzwerkname nicht ohne Weiteres erkennbar und muss den Geräten, die sich an dem WLAN beteiligen sollen, bekannt gemacht werden. Allerdings ist dies nur eine weitere Hürde, es gibt Software, die die SSID trotzdem auslesen kann.

Jede Netzwerkkarte besitzt eine eindeutige *Media Access Control* (*MAC*) Adresse. Die meisten WLAN-Router lassen sich so konfigurieren, dass die Verbindung nur zu Geräten aufgebaut wird, deren MAC-Adresse im Router hinterlegt ist. Allerdings lassen sich MAC-Adressen auch emulieren. Sollte einem potenziellen Angreifer die MAC-Adresse eines für das WLAN berechtigten Gerätes bekannt sein, lässt sich dieser Sicherheitsmechanismus umgehen.

Zusammengenommen lässt sich mit den beschriebenen Massnahmen ein hohes Mass an Sicherheit herstellen.

4.4.2 Werkzeuge

Zur Konfiguration von WLAN Schnittstellen unter Linux gibt es die so genannten *Wireless Tools*. Das zentrale Werkzeug ist `iwconfig` und lehnt sich bzgl. der Bedienung an das Netzwerkkonfigurationswerkzeug *ifconfig* an. Beim Aufruf ohne Parameter zeigt es die aktuellen Einstellungen zu WLAN-Netzwerkschnittstellen an.

```
iwconfig
lo        no wireless extensions.

eth0      no wireless extensions.

wlan0     IEEE 802.11abg  ESSID:"MyPlace"
          Mode:Managed  Frequency:2.412 GHz  Access Point: 00:14:
86:24:14:26
          Bit Rate=54 Mb/s   Tx-Power=15 dBm
          Retry  long limit:7   RTS thr:off    Fragment thr:off
          Power Management:off
          Link Quality=62/70  Signal level=-48 dBm
          Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:
0
          Tx excessive retries:0  Invalid misc:0  Missed beacon:0
```

Angezeigt werden u.a. der Verbindungstyp, die Netzwerkkennung, der Netzwerkmodus, die Verbindungsstärke etc. Mit entsprechenden Parametern beim Aufruf kann die Konfiguration verändert werden. Mit dem Parameter `essid` kann beispielsweise der Netzwerkname gesetzt werden.

```
iwconfig wlan0 essid netzwerkname
```

Mit `iwlist` lassen sich detailliertere Informationen zu einer drahtlosen Netzwerkschnittstelle auflisten, als sie von `iwconfig` angezeigt werden.

```
iwlist wlan0 scanning
wlan0     Scan completed :
          Cell 01 - Address: 00:14:86:24:14:26
                    Channel:1
                    Frequency:2.412 GHz (Channel 1)
                    Quality=61/70  Signal level=-49 dBm
                    Encryption key:on
                    ESSID:"MyPlace"
                    Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 6 Mb/s; 9
 Mb/s
                              11 Mb/s; 12 Mb/s; 18 Mb/s
                    Bit Rates:24 Mb/s; 36 Mb/s; 48 Mb/s; 54 Mb/s
                    Mode:Master
                    Extra:tsf=00000051a0308342
                    Extra: Last beacon: 26684ms ago
                    IE: Unknown: 00074D79506C616365
```

```
                    IE: Unknown: 010882848B0C12961824
                    IE: Unknown: 030101
                    IE: Unknown: 07064E4149010B1B
                    IE: Unknown: 200100
                    IE: Unknown: 2A0100
                    IE: Unknown: 32043048606C
                    IE: Unknown: DD180050F20201018D0002A4400027A4
000042435E0062322F00
                    IE: WPA Version 1
                        Group Cipher : TKIP
                        Pairwise Ciphers (1) : TKIP
                        Authentication Suites (1) : PSK
                    IE: Unknown: DD0900037F01010020FF7F
```

Ausserdem gibt es noch die Werkzeuge `iwspy`, um die Verbindungsqualität zu ermitteln, `iwpriv`, um die Eigenschaften bestimmter Netzwerktreiber zu beeinflussen und `ifrename`, um Netzwerkschnittstellen umzubenennen.

4.5 Infrarot

Mobile Geräte, insbesondere Mobiltelefone, Personal Digital Assistants und Notebooks, sind in den meisten Fällen mit einer *Infrarotschnittstelle* (*engl.: Infrared Interface*) ausgestattet. Es handelt sich dabei um einen optischen Kommunikationsweg, bei dem Daten mit Hilfe von infrarotem Licht übertragen werden. Die Standardisierung eines einheitlichen Kommunikationsprotokolls hat die *Infrared Data Association* (*IrDA*) übernommen, ein Zusammenschluss zahlreicher Unternehmen in einer nicht gewinnorientierten Organisation (http://www.irda.org).

Die Infrarottechnik bietet die Möglichkeit, über eine kurze Entfernung eine Kommunikationsverbindung zwischen zwei Geräten aufzubauen. Da die Daten optisch, also mit Hilfe von Licht übertragen werden, muss zwischen beiden Geräten eine Sichtverbindung bestehen. Kann diese Sichtverbindung nicht aufrechterhalten werden, weil andere Gegenstände zwischen Sender und Empfänger gelangen, wird die Verbindung abgebaut.

Die Version 1.0 des IrDA-Standards unterstützt Datenübertragung mit einer Geschwindigkeit von bis zu 115,2 kBit pro Sekunde (*Serial Infrared* (*SIR*)). IrDA 1.1 erhöht die Übertragungsrate von 1,152 MBit/s bei *Medium Infrared* (*MIR*) über 4 MBit/s bei *Fast Infrared* (*FIR*) bis zu 16 MBit/s bei *Very Fast Infrared* (*VFIR*).

4.5.1 Linux-IrDA

Die Basisfunktionalität für die Verwendung von Infrarot-Hardware ist schon längere Zeit Bestandteil des Linux-Kernels. Software, die für den Zugriff und die Konfiguration nötig ist, wird vom Linux-IrDA Projekt (http://irda.sourceforge.net/) entwickelt.

4.5.2 Werkzeuge

In Form der *irda-utils* beinhaltet Linux-IrDA eine Reihe Hilfsmittel, um Infrarot-Geräte und -Treiber zu konfigurieren.

irattach

Der Befehl `irattach` verbindet die IrDA-Funktionalität aus dem Linux-Kernel mit einem IrDA-Port, über den die Infrarotschnittstelle anschliessend verwendet werden kann. Für die Anbindung kann ein serielles Infrarot-Text-Terminal (*tty*), ein IrDA Gerätetreiber oder eine IrDA Schnittstelle verwendet werden.

Beispielsweise wird mit folgendem Befehl die IrDA-Schicht mit der Schnittstelle `irda0` verbunden und mit dem Parameter `-s` die Erkennung von Infrarot-Geräten in Reichweite gestartet. Dazu muss das Kernel-Modul des Infrarot-Gerätetreibers vorher mit `modprobe` geladen bzw. ein entsprechender `alias` Eintrag in der Datei `/etc/modules.conf` (Kernelversion 2.4) oder `/etc/modprobe.conf` konfiguriert sein.

```
irattach irda0 -s
```

Erkannte Geräte werden in der Datei `/proc/net/irda/discovery` ausgegeben.

```
cat /proc/net/irda/discovery
IrLMP: Discovery log:

nickname: SIEMENS S55, hint: 0xb124, saddr: 0x2273a2b4, daddr: 0x
00000090
```

irdadump

Mit dem Befehl `irdadump` lässt sich der Datenverkehr einer aktivierten Infrarot-Schnittstelle überwachen, ausgeben und protokollieren. Das ist insbesondere für die Konfiguration der Infrarot-Infrastrukur und beim Debugging von Verbindungen hilfreich.

```
irdadump
18:09:03.926294 xid:cmd acac321e > ffffffff S=6 s=* linux-2mmv hi
nt=0400 [ Computer ] (26)
18:09:06.398462 xid:cmd acac321e > ffffffff S=6 s=0 (14)
18:09:06.486455 xid:cmd acac321e > ffffffff S=6 s=1 (14)
18:09:06.574464 xid:cmd acac321e > ffffffff S=6 s=2 (14)
18:09:06.662483 xid:cmd acac321e > ffffffff S=6 s=3 (14)
18:09:06.750477 xid:cmd acac321e > ffffffff S=6 s=4 (14)
18:09:06.833575 xid:rsp acac321e < 00000090 S=6 s=4 SIEMENS S55 h
int=b124 [ PnP Modem Fax IrCOMM IrOBEX ] (28)
18:09:06.838478 xid:cmd acac321e > ffffffff S=6 s=5 (14)
18:09:06.926488 xid:cmd acac321e > ffffffff S=6 s=* linux-2mmv hi
```

```
nt=0400 [ Computer ] (26)
18:09:09.398646 xid:cmd acac321e > ffffffff S=6 s=0 (14)
18:09:09.486644 xid:cmd acac321e > ffffffff S=6 s=1 (14)
18:09:09.574648 xid:cmd acac321e > ffffffff S=6 s=2 (14)
```

irdaping

Mittels `irdaping` können Testdaten an ein erkanntes Infrarot-Gerät gesendet und auf dessen Antwort gewartet werden, falls das entfernte Gerät diese Funktionalität unterstützt. Das ermöglicht, ähnlich wie beim TCP/IP Netzwerkbefehl `ping` die Überprüfung, ob und mit welcher Antwortzeit ein bekanntes Gerät reagiert. Das Ergebnis lässt Rückschlüsse auf die Verbindungsqualität zu. Als Parameter wird die Geräteadresse des entfernten Gerätes benötigt. Diese kann mit `irdadump` oder der Datei `/proc/net/irda/discovery` ermittelt werden.

```
irdaping 0xc214a13b
IrDA ping (0xc214a13b on irda0): 32 bytes
32 bytes from 0xc214a13b: irda_seq=0 time=105.82 ms.
32 bytes from 0xc214a13b: irda_seq=1 time=105.88 ms.
32 bytes from 0xc214a13b: irda_seq=2 time=103.05 ms.
32 bytes from 0xc214a13b: irda_seq=3 time=105.82 ms.
32 bytes from 0xc214a13b: irda_seq=4 time=106.71 ms.
```

Über diese Werkzeuge hinaus, bieten die *irda-utils* noch weitere Programme, z. B. für die Benutzung von Tastaturen, die über die Infrarot-Schnittstelle angebunden werden (`irkbd`) oder ein Hilfsmittel zur Ermittlung des verwendeten Hardware-Chips (`findchip`).

4.5.3 Protokolle

Über eine Infrarotverbindung können diverse Kommunikationsprotokolle eingesetzt werden. Dadurch kann eine Infrarotschnittstelle für unterschiedlichste Zwecke eingesetzt werden.

4.5.3.1 IrCOMM

Mittels *IrCOMM* wird eine serielle Schnittstelle (Abschn. 4.2) über die Infrarot-Hardware emuliert, über den Gerätetreiber `/dev/ircomm0`. Dem Betriebssystem und Anwendungen gegenüber verhält sich der Geräteeintrag wie eine physikalische serielle Schnittstelle und kann auch so angesteuert und verwendet werden. Dadurch können Programme, die für die Kommunikation mittels einer seriellen Schnittstelle entwickelt wurden, ohne Änderung auch über eine Infrarotverbindung Daten austauschen.

4.5.3.2 IrLPT

Über *Infrared Line Printer Protocol* (*IrLPT*) können Drucker angesteuert werden, die mit einer Infrarotschnittstelle ausgestattet sind. Die verwendete Gerätedatei ist `/dev/irlpt0`.

4.5.3.3 IrLAN

Mit dem Protokoll *IrLAN* lässt sich die Infrarot-Schnittstelle als Netzwerkkarte konfigurieren und in ein lokales Netzwerk (engl.: *Local Area Network – LAN*) integrieren. Über das Kernel-Modul `irlan` wird eine Netwerkschnittstelle `irlan0` bereitgestellt, die anschliessend mit `ifconfig` eingerichtet werden kann.

```
modprobe irlan
ifconfig -a
irlan0    Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:4
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

Als Gegenstück wird ein ebenfalls IrLAN-fähiges Gerät, z. B. ein anderer Linux-Rechner benötigt.

4.5.3.4 IrNET

IrNET ist eine Alternative zu *IrLAN*, um das Protokoll *TCP/IP* über eine Infrarot-Verbindung zu übertragen. IrNET wurde von der Firma Microsoft eingeführt und ist nicht Bestandteil der Spezifikation von IrDA.

Im Gegensatz zu IrLAN verwendet IrNET das so genannte *Point To Point Protocol* (*PPP*), das ursprünglich für den Verbindungsaufbau über Wählleitungen eingeführt wurde. Es wird also auf einen etablierten und bewährten Standard gesetzt. PPP-Verbindungen sind nicht auf Internet-Protokolle beschränkt, was wiederum die Flexibilität erhöht und Einsatzmöglichkeiten erweitert. Die Spezifikation beinhaltet u.A. Möglichkeiten zur Authentifizierung, Verschlüsselung und Datenkompression.

4.5.3.5 IrOBEX

IrOBEX bzw. *OBEX* steht für *InfraRed OBject EXchange* und dient der Übertragung von Datenobjekten zwischen zwei Infrarot-Geräten. Dieses Protokoll wird z. B. zur Synchronisation von PDAs oder Mobiltelefonen eingesetzt.

4.6 Bluetooth

Bluetooth ist ein internationaler Standard für Kurzstreckenfunknetzwerke (http://german.bluetooth.com), dessen Entwicklung von der Firma Ericsson initiiert wurde, inzwischen aber von einer Gruppe von Unternehmen vorangetrieben wird, die sich zur *Bluetooth Special Interest Group* (*SIG*) zusammengeschlossen haben. Ziel des Standards ist es, Kabelverbindungen zwischen mobilen Geräten überflüssig zu machen und das auf sichere, zuverlässige, energiesparende und kostengünstige Weise. Bluetooth eignet sich daher hervorragend für den Einsatz im Pervasive Computing.

Mehrere Bluetooth-fähige Geräte schließen sich spontan (ad hoc) zu einem so genannten *Piconet* zusammen, dem bis zu sieben Geräte angehören können, wobei ein Gerät mehreren Piconets angehören kann. Piconets werden automatisch und dynamisch aufgebaut, sobald sich mindestens zwei Geräte in Sende- bzw. Empfangsreichweite befinden. Eines der beteiligten Geräte übernimmt die Steuerung des Piconets, und wird damit der so genannte *Master*, während sich jedes weitere Gerät als *Slave* in das Netz eingliedert.

Die Reichweite von Bluetooth hängt von der eingesetzten Hardware ab und reicht von maximal 1 m (Class 1), über maximal 10 m (Class 2) bis zu maximal 100m (Class 3). Die verwendete Funkfrequenz von 2,4 GHz ist in den meisten Ländern lizenzfrei, so dass der Einsatz von Bluetooth, bzw. die Nutzung der Übertragungsfrequenz kostenlos ist. Die Spezifikation von Bluetooth erlaubt die gleichzeitige Übertragung von Sprache und Daten über die selbe Funkverbindung.

Bluetooth-fähige Geräte werden abhängig von ihrem Einsatzzweck in so genannte *Profile* eingeteilt. Das Basisprofil *Generic Access Profile* (*GAP*) legt die minimale Funktionalität fest, die jedes Bluetooth-Gerät unterstützen muss und auf das alle anderen Profile aufbauen. Ein Bluetooth-Gerät, das keine anwendungsspezifischen weiteren Profile unterstützt muss dieses Profil beherrschen, um minimale Verbindungsfunktionen anzubieten.

Weitere Profile sind z. B. *Dial-up Networking Profile* (*DUN*) zur Einwahl in ein anderes System, wobei das eine Bluetooth-Gerät als Modem verwendet wird, *Fax Profile* (*FAX*) für das Versenden von Fax-Nachrichten, *Headset Profile* (*HSP*) für die Verbindung zu Headsets von Mobiltelefonen und *Personal Area Networking Profile* (*PAN*), um ein spontanes Netzwerk mit mehreren Bluetooth-Geräten aufzubauen. Eine vollständig Übersicht über die definierten Profile findet sich auf der Webseite der Bluetooth SIG.

4.6.1 Sicherheit

Sicherheit und Zugriffsschutz wurde bei der Entwicklung der Spezifikation von Bluetooth von vornherein berücksichtigt. Dabei sind prinzipiell drei Sicherheitsstufen vorgesehen:

Unsicher: In diesem Fall besteht keine Sicherheit, Daten werden unverschlüsselt und ohne Zugriffsschutz übertragen.

Dienstesicherheit: Diese Ebene wird in drei Stufen unterteilt:

- Dienste, die von allen Geräten genutzt werden können.
- Dienste, die eine Authentifizierung erfordern.
- Dienste, die sowohl Authentifizierung als auch Authorisierung erfordern.

Auf dieser Sicherheitsstufe entscheiden die Anwendungen, welche von Bluetooth angebotenen Sicherheitsmöglichkeiten genutzt werden sollen.

Verbindungssicherheit: Verbindungssicherheit bedeutet, dass eine Bluetooth-Verbindung nur über eine verschlüsselte Authentifizierung zustande kommt, optional können auch die übertragenen Daten verschlüsselt werden.

Damit Bluetooth-Geräte miteinander kommunizieren können, müssen sie beim ersten Verbindungsaufbau eine gegenseitige Vertrauensbeziehung aufbauen, das so genannte *Pairing*. Dazu muss für beide beteiligte Geräte eine gemeinsame mindestens vierstellige Identifikationsnummer vergeben werden, die auf beiden Geräten einzugeben, bzw. zu hinterlegen ist. Auf diese Weise wird sichergestellt, dass nur Geräte miteinander kommunizieren, für die dies explizit vorgesehen ist.

4.6.2 BlueZ

Für die Unterstützung von Bluetooth unter Linux hat sich das Projekt BlueZ (http://www.bluez.org/) durchgesetzt, welches ursprünglich von der Firma Qualcomm gestartet wurde. Der aus BlueZ hervorgegangene für den Linux-Kernel relevante Code ist inzwischen auch Bestandteil der offiziellen Linux Kernel-Quellen.

Um die im Kernel verfügbare Funktionalität nutzen zu können, müssen die Bluetooth Bibliotheken `bluez-libs` und Werkzeuge `bluez-utils` des BlueZ-Projektes installiert sein.

4.6.2.1 BlueZ Bibliothek

Die BlueZ Bibliothek `bluez-libs` ermöglicht den Zugriff auf die im Linux Kernel vorhandene Bluetooth-Funktionalität unabhängig von der verwendeten Hardware.

Host Controller Interface (HCI)

Die so genannte Hostcontroller-Schnittstelle (engl.: *Host Controller Interface – HCI*) wird von der unterliegenden verwendeten Hardware abstrahiert. Über sie

können Befehle an die Steuereinheiten bzw. Geräte geschickt und auf entsprechende Konfigurationsparameter zugegriffen werden.

Logical Link Control and Adaptation Layer Protocol (L2CAP)

Diese Netwerkschicht baut auf HCI auf und erlaubt verbindungsorientierte und verbindungslose logische Dienste über Bluetooth. Somit können logische Verbindungen zwischen Anwendungen aufgebaut und Daten über das *L2CAP*-Protokoll übertragen werden.

Radio Frequency Communication (RFCOMM)

Die Kommunikationsschicht *RFCOMM* emuliert eine serielle Schnittstelle auf Basis der L2CAP Schicht. Damit ist es möglich, existierende Kommunikationssoftware zu verwenden, die andere Geräte über eine (bisher kabelgebundene) serielle Schnittstelle (Abschn. 4.2) ansteuert.

4.6.2.2 Bluetooth Network Encapsulation Protocol (BNEP)

Das Bluetooth Network Encapsulation Protocol kapselt diverse existierende Netzwerkprotokolle und transportiert sie über eine L2CAP Bluetooth-Verbindung. Diese Vorgehensweise ermöglicht den spontanen Aufbau eines Netzwerkes mit mehreren Bluetooth-Geräten anhand des so genannten *Personal Area Network* (*PAN*) Profiles der Bluetooth-Spezifikation.

Service Discovery Protocol (SDP)

Die Suche nach von Bluetooth-Geräten angebotenen Diensten wird durch das Protokoll zum Auffinden von Bluetooth-Diensten (engl.: *Service Discovery Protocol – SDP*) ermöglicht.

4.6.2.3 Werkzeuge

Mit den BlueZ Werkzeugen kann die Bluetooth-Funktionalität konfiguriert und verwendet werden.

hciconfig

Für die Konfiguration und Verwaltung von Bluetooth-Geräten eignet sich das Werkzeug `hciconfig`. Der Aufruf ohne Parameter zeigt den Status aller konfigurierten Bluetooth-Geräte an.

```
hciconfig
hci0:   Type: USB
        BD Address: 00:20:E0:77:C8:F1 ACL MTU: 192:8  SCO MTU: 64
:8
```

```
        UP RUNNING PSCAN
        RX bytes:4678 acl:12 sco:0 events:126 errors:0
        TX bytes:2634 acl:10 sco:0 commands:88 errors:0
```

Die Funktionsweise von `hciconfig` ist an die Verwaltung von Netzwerkschnittstellen mit `ifconfig` angelehnt. Der Parameter `-a` gibt detaillierte Informationen über die verfügbaren Geräte an.

```
hciconfig -a
hci0:   Type: USB
        BD Address: 00:20:E0:77:C8:F1 ACL MTU: 192:8 SCO MTU: 64:
8
        UP RUNNING PSCAN
        RX bytes:4678 acl:12 sco:0 events:126 errors:0
        TX bytes:2634 acl:10 sco:0 commands:88 errors:0
        Features: 0xff 0xff 0x0f 0x00 0x00 0x00 0x00 0x00
        Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
        Link policy: RSWITCH HOLD SNIFF PARK
        Link mode: SLAVE ACCEPT
        Name: 'BlueZ linux-2mmv (0)'
        Class: 0x00010c
        Service Classes: Unspecified
        Device Class: Computer, Laptop
        HCI Ver: 1.1 (0x1) HCI Rev: 0x222 LMP Ver: 1.1 (0x1) LMP
Subver: 0x222
        Manufacturer: Cambridge Silicon Radio (10)
```

hcid

Der HCI Dämon steuert als Dienst den Zugriff auf die Hostcontroller- Schnittstelle, um die Gerätekonfiguration dauerhaft zu speichern. Konfiguriert wird dieser Dienst über die Datei `/etc/bluetooth/hcid.conf`. Hier wird u.a. definiert, unter welchem Namen das Bluetooth-Gerät angesprochen wird, um welche Geräteklasse es sich handelt, sowie welche Sicherheitsmechanismen verwendet werden sollen. Nach einer Änderung in der Datei muss der Dienst erneut gestartet werden, damit die Konfiguration wirksam wird.

```
/etc/init.d/bluetooth restart
```

hcitool

Mit dem Werkzeug `hcitool` können grundlegende Funktionen wie die Suche nach erreichbaren Geräten oder das Aufbauen von Verbindungen durchgeführt werden.

Beispielsweise gibt der folgende Befehl alle in Reichweite sichtbaren Bluetooth-Geräte aus.

```
hcitool scan
Scanning ...
        00:01:E3:71:75:CB       C2C
```

sdptool

`sdptool` hilft sowohl dabei, erreich- und verfügbare Dienste in der Umgebung zu finden, als auch über den Dienst `sdpd` selbst angebotene Diente zu verwalten. Mit dem Befehl `browse` wird nach verfügbaren Diensten gesucht.

```
sdptool browse
Inquiring ...
Browsing 00:01:E3:71:75:CB ...
Service Name: SerialPort
Service RecHandle: 0x11101
Service Class ID List:
  "Serial Port" (0x1101)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 1
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
Profile Descriptor List:
  "Serial Port" (0x1101)
    Version: 0x0100

...
```

Um die Ergebnisliste einzugrenzen kann auch nur ein bestimmtes Gerät abgefragt werden, z. B. `sdptool browse 00:01:E3:71:75:CB`.

rfcomm

Mittels `rfcomm` werden emulierte serielle Verbindungen über Bluetooth konfiguriert und aufgebaut. Zur temporären Verbindung zu einem Bluetooth- Gerät über `rfcomm` dient der Befehl `connect`. Beispielsweise kann zu dem entfernten Gerät (z. B. ein Mobiltelefon) mit der Bluetooth-Adresse `00:01:E3:71:75:CB` eine Verbindung hergestellt werden.

```
rfcomm connect 0 00:01:E3:71:75:CB
Connected /dev/rfcomm0 to 00:01:E3:71:75:CB on channel 1
Press CTRL-C for hangup
```

Dadurch wird dynamisch die Gerätedatei `/dev/rfcomm0` erzeugt und kann so lange wie eine herkömmliche serielle Schnittstelle verwendet werden, bis die Verbindung mit der Tastenkombination `Strg-C` wieder beendet wird.

Für wiederkehrende Verbindungen, die regelmässig genutzt werden, bietet es sich an, das Gerät fest an das logische Gerät `/dev/rfcomm0` zu binden.

```
rfcomm bind 0 00:01:E3:71:75:CB
```

Nach diesem Aufruf steht die Gerätedatei `/dev/rfcomm0` zur Verfügung und kann wie eine serielle Schnittstelle verwendet werden. Beim ersten Zugriff auf dieses Gerät wird dann das *Pairing* durchgeführt, d.h. auf beiden Seiten wird die PIN für die Verbindung angefordert. Das Lösen der Verbindung erfolgt entsprechend mit dem `release` Befehl.

```
rfcomm release 0
```

Über die Datei `/etc/bluetooth/rfcomm.conf` erfolgt eine permanente Konfiguration. Hier kann z. B. spezifiziert werden, dass das im Beispiel verwendete Gerät automatisch an die Schnittstelle `/dev/rfcomm4` gebunden werden soll.

```
rfcomm4 {
       # Automatically bind the device at startup
       bind yes;

       # Bluetooth address of the device
       device 00:01:E3:71:75:CB;

       # RFCOMM channel for the connection
       channel 1;

       # Description of the connection
       comment "Mobiltelefon";
}
```

Die definierten Schnittstellen können nun an die entsprechenden Geräte gekoppelt werden.

```
rfcomm bind all
```

Üblicherweise werden die konfigurierten Schnittstellen beim Start von Bluetooth über das Skript `/etc/init.d/bluetooth` gebunden. Welche Verbindungen bestehen, gibt der Parameter `-a` aus.

```
rfcomm -a
rfcomm4: 00:01:E3:71:75:CB channel 1 clean
```

pand

Das Werkzeug `pand` dient dazu, spontane aus Bluetooth-Geräten bestehende Netze in der unmittelbaren persönlichen Umgebung aufzubauen (engl.: *Personal Area Networks – PAN*). Dabei wird über den Dienst entweder das lokale Gerät als Netzwerkverbindungspunkt (engl.: *Network Access Point – NAP*) konfiguriert, zu dem sich andere Bluetooth-Geräte verbinden können, oder es wird eine Verbindung zu einem anderen Verbindungspunkt aufgebaut.

Um als NAP zu fungieren, muss der Dienst mittels des Parameters `--listen` bzw. `-s` angewiesen werden, auf eingehende Verbindungen zu warten, im *Piconet* die Hauptrolle zu spielen (`-M` bzw. `--master`), sowie als Zugriffspunkt des PANs zu agieren (`--role NAP` oder `-r NAP`).

```
pand -s -M -r NAP
```

Der Zugriff auf einen NAP funktioniert mit der Option `-c` bzw. `--connect`.

```
pand -c 00:20:E0:77:C8:F1
```

Damit wird die Verbindung aufgebaut und auf beiden Seiten steht eine dynamisch erstellte Netzwerkschnittstelle `/dev/bnep0` zur Konfiguration bereit.

```
ifconfig -a
bnep0     Link encap:Ethernet  HWaddr 00:20:E0:77:C8:F1
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

Über diese Schnittstelle kann nun z. B. das Internet-Protokoll TCP/IP konfiguriert werden.

Der aktuelle Verbindungsstand des Dienstes kann über die funktionsgleichen Optionen `--show`, `--listen` oder `-l` ermittelt werden.

```
pand -l
bnep0 00:19:7D:E7:89:F3 NAP
```

Mit der Option `--search`/`-Q` kann automatisch nach einem anderen Gerät gesucht und eine Verbindung aufgebaut werden.

4.7 Mobilfunk

Mobile Telefonie ist der vermutlich grösste Wachstumsmarkt für das Pervasive Linux. Als Netzwerkstandard ist das *Global System for Mobile Communications* (*GSM*) am verbreitetsten. Für Datenkommunikation hat sich *General Packet Radio Service* (*GPRS*) durchgesetzt. Inzwischen bekommt das *Universal Mobile Telecommunications System* (*UMTS*) zunehmend Bedeutung, da in diesem Netz deutlich höhere Datenübertragungsgeschwindigkeiten möglich sind.

Für den Einsatz von Linux auf Mobiltelefonen existieren eine Reihe speziell angepasster Umgebungen.

4.7.1 Openmoko

Das ursprüngliche Ziel des Projektes *Openmoko* war es, eine freie und offene Softwareplattform für Mobiltelefone bereitzustellen. Aus diesem Vorsatz ging eine neue Firma gleichen Namens (http://www.openmoko.com) hervor, die es sich zum Ziel gesetzt hat, offene Produkte für den mobilen Einsatz herzustellen. Das erste Mobiltelefon wurde in der Entwicklerversion im Juli 2007 unter dem Namen *Neo 1973* ausgeliefert. Die komplette Hard- und Softwarespezifikation ist im Internet offengelegt. Unter der Adresse http://www.openmoko.org ist die Entwicklergemeinde des Projektes OpenMoko im Internet zu finden.

Der Softwareanteil des Projektes Openmoko besteht aus einem Linuxkernel, den für die mobile Kommunikation benötigten Gerätetreibern, einem X-Server sowie Anwendungsprogrammen und Bibliotheken für den mobilen Einsatz. Dieser Satz an Softwarekomponenten kann für weitere Pervasive Computing Geräte über die Openmoko Hardware hinaus übersetzt werden.

Als Bootloader auf der Openmoko Hardware wird U-Boot (Abschn. 3.1.1) verwendet. Anstatt der Openmoko-Distribution können alternative Umgebungen ebenfalls auf den Geräten von Openmoko installiert werden.

4.7.2 Android

Android (http://code.google.com/android/) ist die Softwareplattform der *Open Handset Alliance* (http://www.openhandsetalliance.com/) und wird hauptsächlich von der Firma Google entwickelt. Bei Android handelt es sich um eine Umgebung für Anwendungen, die in der Programmiersprache *Java* geschrieben sind und auf mobilen Geräten eingesetzt werden. Android-Programme sind auf der Dalvik virtuellen Maschine lauffähig, die speziell für den Einsatz im Pervasive Computing entwickelt wurde.

Die Android-Plattform basiert auf einem Linux-Kernel als Betriebssystem.

4.7.3 GPE Phone Edition

Die *GPE Phone Edition* setzt auf GTK und GPE für die Entwicklung von Software für mobile Endgeräte, insbesondere Mobiltelefone. Zu finden ist das Open Source Projekt unter der Internetadresse http://gpephone.linuxtogo.org/.

4.7.4 LiMo Foundation

In der *LiMo Foundation* haben sich Hersteller von mobilen Kommunikationsgeräten zusammengeschlossen, um eine offene Linuxplattform für mobile Geräte, vor allem Mobiltelefone, bereitzustellen. Der LiMo Foundation (http://www.limofoundation.org) geht es darum, eine marktfähige Plattform für den mobilen Einsatz von Geräten, Anwendungen und Dienstleistungen bereitzustellen.

Eine weitere Industrieinitiative, verlässliche Standards im Bereich der Mobiltelefonie unter Linux zu definieren, das *Linux Phone Standards Forum* (*LiPS*), zu finden unter http://www.lipsforum.org wurde mittlerweile in die LiMo Foundation integriert.

4.7.5 phoneME

Das Open Source Projekt *phoneME* (https://phoneme.dev.java.net) ist eine Initiative der Firmen SUN Microsystems, O'Reilly und CollabNet, um die Verbreitung der Java Plattform in der so genannten Micro Edition zu fördern.

Kapitel 5
Werkzeuge

Die Werkzeugpalette für die Software-Entwicklung für Pervasive Linux-Systeme stammt grösstenteils aus der Entwicklung für herkömmliche Linux-Systeme. Allerdings gibt es auch einige Werkzeuge und Programme, die speziell für den Einsatz von Pervasive Linux entwickelt oder erweitert wurden.

Dieses Kapitel stellt eine Zusammenstellung von Werkzeugen vor, die die Entwicklung von Software für Pervasive Linux relevant sind und diese erleichtern oder erst ermöglichen.

5.1 GCC – GNU Compiler Collection

Wer sich mit Linux als Entwickler beschäftigt, stösst unweigerlich auf die GCC, die GNU Compiler Collection der Free Software Foundation. Hervorgegangen aus einem freien C-Compiler beinhaltet die GCC heute den meistgenutzten Compiler der Sprachen C und C++ für die Entwicklung unter Linux und vielen anderen Betriebssystemen. Man muss kein Softwareentwickler sein, um mit der GCC konfrontiert zu werden, die Skripte zur Erstellung eines angepassten Linux-Kernel zum Beispiel setzen die GCC und deren C-Compiler voraus, um einen neuen Kernel zu übersetzen. Auch viele Programme, die nur als Quellcode ausgeliefert werden, werden im Zuge der Installation zuerst übersetzt.

5.1.1 Das Kreuz mit dem Cross Compiler

Bei der Softwareentwicklung für Desktops entspricht normalerweise die Umgebung, die für die Entwicklungsaktivitäten eingesetzt wird, derjenigen, in der die fertiggestellte Software schließlich ausgeführt werden soll. Das erleichtert die Entwicklung, da alle zur Erstellung der Software notwendigen Schritte (Kodierung, Übersetzen, Ausführen und Test) auf ein und demselbem System durchgeführt werden können.

Im Gegensatz dazu stehen auf eingebetteten Systemen oft nicht ausreichend Ressourcen zur Verfügung, um alle Entwicklungsschritte direkt in der Zielumgebung zu durchlaufen. Eventuell lassen sich für die Entwicklung leistungsfähigere

C. Czernohous, *Pervasive Linux*, X.systems.press,
DOI 10.1007/978-3-540-68426-8_5, © Springer-Verlag Berlin Heidelberg 2012

Umgebungen mit mehr Speicherplatz oder schnelleren Prozessoren konfigurieren, so dass das Übersetzen und Ausführen direkt auf einem System der Zielarchitektur durchgeführt werden kann. Im Folgenden wird unterschieden zwischen Entwicklungssystem bzw. -umgebung als dem Ort, auf dem ausführbarer Code erzeugt wird, und Zielsystem bzw. -umgebung, wo der erzeugte Code schließlich ausführbar sein soll.

Eine praktikable Vorgehensweise, insbesondere wenn die Übertragung von Daten auf das Zielsystem umständlich und zeitaufwendig ist, ist es, die Software zuerst auf dem Entwicklungssystem fertigzustellen und erst bei Erreichen einer akzeptablen Stabilität mit dem Testen auf dem Zielsystem zu beginnen. Diese Vorgehensweise wird erleichtert, je ähnlicher sich Entwicklungs- und Zielsystem sind, also z. B. beide Systeme Linuxumgebungen sind.

Damit der auf dem Entwicklungssystem erzeugte Code in der Zielumgebung ausführbar ist, muss im Fall von nicht interpretierten, also übersetzten, Programmiersprachen beim Übersetzungsvorgang auf dem Entwicklungssystem Binärcode für die Hardwarearchitektur der Zielumgebung entstehen, die sich eventuell von der Hardwarearchitektur der Entwicklungsumgebung unterscheidet. Dieser Vorgang wird als Cross Compilation bezeichnet.

Und selbst wenn der komplette Übersetzungsvorgang auf dem Zielsystem durchgeführt werden soll, müssen in den meisten Fällen die dazu notwendigen Werkzeuge einmal für die Zielplattform erstellt werden, da diese selten in einer direkt auf der Zielplattform ausführbaren Version zur Verfügung stehen.

5.1.2 Toolchain

Für die Generierung von ausführbarem Code wird der ursprüngliche Quellcode in einer Reihe von Schritten in diverse Zwischenformate überführt, bis schließlich als Endprodukt das lauffähige Programm entsteht. Die für die Erstellung der Zwischenprodukte und des ausführbaren Codes benötigten Programme werden als Werkzeugkette (engl. Toolchain) bezeichnet. Im weiteren Text wird der englische Begriff Toolchain benutzt.

Eine Toolchain beinhaltet die für den Schritt der Umwandlung von Quell- in ausführbaren Code benötigten Hilfsmittel wie Assembler, Compiler oder Linker.

Insbesondere für das Erstellen von Code zur Ausführung auf einer Zielplattform, die sich bezüglich der Hardwarearchitektur von der Entwicklungsplattform unterscheidet, was bei eingebetteten Systemen die Regel ist, wird eine spezielle Toolchain benötigt, die Cross Compilation für die Zielplattform unterstützt.

Der einfachste Weg, an eine Toolchain zu gelangen, die Cross Compilation für die Zielplattform unterstützt, ist die Benutzung einer bereits fertig erstellten. Je verbreiteter die Zielplattform ist, desto wahrscheinlicher ist es, auf den entsprechenden Seiten im Internet fündig zu werden, die sich mit Linux für die gewünschte Plattform befassen. Fertige Toolchains haben allerdings den Nachteil, dass sie oft veraltete Versionen der Werkzeuge verwenden und es nicht immer nachvollziehbar

ist, welche Änderungen an der Toolchain oder dem Linux-Kernel vorgenommen wurden.

Falls die angebotenen Toolchains den Anforderungen nicht genügen oder es für die gewünschte Zielplattform keine vorgefertigte Toolchain gibt, bleibt nur die Zusammenstellung einer auf diesen Anwendungsfall zugeschnittenen Toolchain.

Die Erstellung einer Toolchain für Cross Compilation auf Basis der GNU Werkzeuge, ist ein zeitaufwändiger und fehleranfälliger Prozess. Bevor man sich entschliesst, eine eigene Toolchain zu erstellen, sollten Alternativmöglichkeiten geprüft werden, an eine solche zu gelangen. Die diversen Bestandteile der Toolchain werden unabhängig von einander entwickelt und nicht jede Version einer Komponente arbeitet problemlos mit der Version einer anderen Komponente zusammen. Es braucht meistens zahlreiche Anläufe und Versuche, bis man zu einer vollständigen funktionierenden Toolchain kommt.

Im folgenden wird der Aufbau einer GNU Toolchain für die ARM Plattform eines iPaq beschrieben. Ziel ist eine Toolchain, mit der in einer x86-basierten Umgebung Code erstellt werden kann, der auf einem ARM Linuxsystem ausführbar ist. Die Vorgehensweise ist für andere Zielplattformen entsprechend.

Für die Zusammenstellung einer Toolchain werden die benötigten Komponenten mit Hilfe einer angepassten Konfiguration aus deren Quellen übersetzt. Das setzt voraus, dass auf der Entwicklungsumgebung bereits eine einsatzfähige GCC Toolchain installiert ist, die auf der Entwicklungsplattform lauffähigen Code erzeugt. Jede Linuxdistribution bringt eine solche Toolchain mit, ggf. muss sie vom Distributionsmedium nachinstalliert werden. Man muss also zwischen zwei Toolchains unterscheiden, derjenigen, mit der später Programme für die Zielplattform erzeugt werden sollen und derjenigen, mit der erstere selbst übersetzt wird. Die Vorgehensweise, zu in der Zielumgebung ausführbarem Code zu gelangen, ist demnach zweistufig.

5.1.2.1 Schritte der GNU Toolchain-Erstellung

Die in der Entwicklungsumgebung installierte Toolchain kann normalerweise nur ausführbaren Code für die gleiche Plattform erzeugen. Zum Beispiel lassen sich in einer x86 basierten Umgebung Programme übersetzen, die in einer Umgebung auf Basis der Intel x86-Architekur ausführbar sind. So lassen sich auf einem System mit i586-Prozessor (Pentium) Programme übersetzen, die auf einem i386- oder AMD KII-basierten System ausführbar sind und umgekehrt.

Teile der GNU Toolchain lassen sich so konfigurieren und übersetzen, dass die Übersetzungsprodukte auf diversen Plattformen lauffähig sind. Mit diesem Teil der Toolchain können dann die übrigen Bestandteile erstellt werden, die in auf der Zielplattform lauffähiger Form vorliegen müssen. Im ersten Schritt muss also eine Teil-Toolchain erzeugt werden, die Quellcode übersetzen kann, der in übersetzter Form auf der Zielplattform ausführbar ist. Dazu müssen Teile der Toolchain in einer speziellen Konfiguration so übersetzt werden, dass ihre Übersetzungsergebnisse auf der Zielplattform lauffähig sind, die Toolchain selbst jedoch auf der Entwicklungsplattform ausgeführt werden kann.

Erst im zweiten Schritt kann diese Toolchain dann vervollständigt werden und Quellcode übersetzen, der direkt auf der Zielplattform ausführbar ist.

Soll die Toolchain selbst in der Zielumgebung einsatzfähig sein, so muss sie mit der im vorherigen Schritt erstellten Toolchain, wie alle Programme für die Zielplattform, erneut übersetzt werden, damit sie auf dieser ausgeführt werden kann. In diesem Fall kommt also noch ein dritter Schritt hinzu, der die Toolchain für die Zielumgebung übersetzt und erst dann kann der eigentliche Quellcode für die zu erstellende Anwendung übersetzt werden.

Wichtig für die Erstellung einer Toolchain für die Zielplattform ist, dass der verwendete Dateisystemtyp so genannte weiche Verweise (engl.: Soft Links) unterstützt. Mit Soft Links ist es möglich, einen Dateieintrag im Dateisystem auf eine andere Datei oder einen anderen Soft Link verweisen zu lassen.

5.1.2.2 Bestandteile der GNU Toolchain

Für den Aufbau einer Toolchain auf Basis der GCC, werden folgende Komponenten benötigt:

GNU binutils:	Die binutils spielen als Sammlung von Werzeugen wie Assembler und Binder (Linker) beim Erzeugen und Zusammensetzen der diversen Zwischenprodukte im Übersetzungsprozess eine zentrale Rolle.
GNU GCC:	Die Compiler-Sammlung als Kern des Übersetzungsvorgangs setzt Quellcodeteile in Objektcodeteile um, die dann von den binutils zu einer ausführbaren Einheit verbunden werden.
Linux Kernel-Headers:	Die Header-Dateien werden für das Übersetzen des Compilers und der Sprachbibliothek benötigt. Sie definieren die Funktionsadressen für Aufrufe der Kernel-Funktionen.
Eine Sprachbibliothek:	Sie beinhaltet Funktionalität, die der GCC Compiler für die Unterstützung weiterer Sprachen ausser C benötigt. In der Basisversion kann der GCC Compiler ausschliesslich C-Programme übersetzen. Die verbreitetste Sprachbibliothek für die GCC ist glibc.

Darüber hinaus werden je nach Anwendungsfall und Entwicklungsstand der Werkzeugquellen weitere Komponenten benötigt, wie z. B. spezielle vom Compiler vorausgesetzte Bibliotheken oder sogenannte Patches, die Fehler im Quellcode beheben oder zusätzliche Funktionalität in diesen einfügen. Diese Patches sind abhängig von den Versionsständen der beteiligten Komponenten der Toolchain.

Die meisten dieser Komponenten werden mehr oder weniger unabhängig voneinander entwickelt, so dass es zu Inkompatibilitäten zwischen den Versionsständen kommen kann. Um zu einer durchgängig kompatiblen Zusammenstellung zu gelangen, bleibt nichts anderes übrig, als diverse Versionsstände miteinander auszuprobieren. Schneller ist es, im Internet auf den entsprechenden Webseiten oder

Newsgroups nach bekannt funktionsfähigen Versionskombinationen und den benötigten Komponenten zu suchen. Für das folgende Beispiel werden diese Komponenten verwendet:

binutils-2.18:	Die GNU Binärwerkzeuge
gcc-3.4.6:	Die GNU Compiler Collection
glibc-2.3.6:	Die GNU Sprachbibliothek
glibc-linuxthreads:	Die Unterstützung von Multi-Thread-Funktionalität wird unter anderem vom C++ Teil der glibc benötigt.

Die Version 2.4 und höher von glibc sind für Linux Kernel-Versionen 2.6 und höher angepasst. Für diese Kernel-Versionen wird für die Unterstützung zusätzlicher Plattformen über die x86 basierten Prozessoren hinaus ein weiteres Paket mit den Quellen (glibc-ports) benötigt. Allerdings wird das „ports“ Paket nicht offiziell von den Entwicklern der glibc gewartet.

5.1.2.3 Entpacken der Quellen

Im Folgenden wird als Quellenverzeichnis

```
/tmp/arm-toolchain/
```

verwendet. In dieses Verzeichnis werden die Quellen von binutils, gcc und glibc in entsprechende Verzeichnisse entpackt, und es dient auch als Ausgangspunkt für die Navigation durch den Verzeichnisbaum für Verzeichnispfade, die in relativer Form angegeben werden.

Die folgenden Schritte auf der Kommandozeile entpacken die Quellen der Toolchain und die Thread-Funktionalität der glibc.

```
cd /tmp/arm-toolchain/
gunzip binutils-2.18.tar.gz
gunzip gcc-3.4.6.tar.gz
gunzip glibc-2.3.6.tar.gz
gunzip glibc-linuxthreads-2.3.6.tar.gz
tar -xvf binutils-2.18.tar
tar -xvf gcc-3.4.6.tar
tar -xvf glibc-2.3.6.tar
cd glibc-2.3.6
tar -xvf ../glibc-linuxthreads-2.3.6.tar
```

5.1.2.4 Erstellen der Build-Verzeichnisse

In den Build-Verzeichnissen werden die Konfigurationsdateien für die Übersetzungsvorgänge der Bestandteile der Toolchain sowie die fertigen Ergebnisse abgelegt. Es empfiehlt sich, dafür nicht direkt die Verzeichnisse zu benutzen, in die die Quellen der Toolchain-Bestandteile entpackt wurden, sondern separate Verzeichnisse anzulegen, damit die Übersetzungsergebnisse klar von deren Quellen getrennt sind. Hier wird das Hauptverzeichnis (in diesem Fall /tmp/arm-toolchain)

verwendet, in dem sich auch die Unterverzeichnisse mit den entpackten Quellen befinden.

```
mkdir binutils
mkdir gcc
mkdir glibc
```

Für alle Bestandteile der GNU Toolchain gilt, dass sie ein Konfigurationsskript mit dem Namen „configure“ enthalten, mit dem die Quellen für den anschliessenden Übersetzungsprozess vorbereitet werden. Mit dem Parameter „–help“ gibt das Skript die Liste möglicher Konfigurationsparameter aus.

5.1.2.5 Übersetzen von binutils

Ohne das Paket binutils funktioniert der GCC Compiler nicht, es beinhalt essenzielle Hilfsmittel für die Generierung von ausführbarem Code. Deshalb wird es als erstes übersetzt. Da die Programme der binutils den auf der Zielplattform ausführbaren Code bearbeiten, muss eine Version erzeugt werden, die diese Aktivitäten so ausführt, wie von der Zielplattform vorausgesetzt werden. Vor dem Übersetzungsvorgang müssen aber noch ein paar Entscheidungen getroffen werden und dem Konfigurationsskript mittels der folgenden Parameter mitgeteilt werden. Zumindest der Parameter `target` ist für ein späteres Übersetzen für eine andere Plattform nötig. Die übrigen Konfigurationsparameter ermittelt das Skript durch entsprechende Tests bzw. sind mit Standardwerten vorbelegt, die in diesem Beispiel übernommen werden:

target: Die Zielplattform, auf der Programme, die mit der zu erstellenden Toolchain erzeugt werden, ausführbar sein sollen. In diesem Fall ist es die Plattform arm-linux. Wird dieser Parameter nicht angegeben, wird der Übersetzungsvorgang für die Plattform konfiguriert, auf der das Skript ausgeführt wird, in dem in diesem Kapitel vorgestellten Beispiel ist das eine x86-Plattform.

build: Mit diesem Wert wird spezifiziert, auf welcher Plattform der Erstellungsvorgang erfolgen wird. Das Skript für die Konfiguration der Übersetzungsregeln der GNU binutils ist weitestgehend plattformunabhängig, d.h. unverändert auf einer großen Zahl Hardwareplattformen einsetzbar. Durch entsprechende Tests des configure Skripts wird in den meisten Fällen ein korrekter Wert ermittelt.

host: Die Entwicklungsumgebungsplattform, auf der die Toolchain genutzt werden soll. Dieses Beispiel wird für eine generische x86-Architektur erstellt, also ist der anzugebende Wert i386. Wird für den Parameter –host kein Wert angegeben, wird derselbe Wert verwendet, der für den Parameter build angegeben oder ermittelt wurde.

prefix: Das Verzeichnis in dem die Toolchain auf dem Entwicklungssystem installiert werden soll. Dieses Verzeichnis kann später nicht mehr geändert werden, ausser durch Neuübersetzen der binutils Quellen. Der Standardwert, den das Skript annimmt, wenn der Parameter nicht angegeben wird, ist /usr/local. In diesem Beispiel kann die Angabe also entfallen.

Das Hauptergebnis des Konfigurationsskripts ist die Steuerdatei `Makefile`, die die benötigten Übersetzungsschritte beinhaltet und von dem Werkzeug `make` ausgelesen wird. Mit dem Aufruf von „make“ wird das Paket binutils gebaut. Als letzter Schritt erfolgt die Installation im Verzeichnis /usr/local, der von einem Benutzer mit root Berechtigung ausgeführt werden muss.

```
cd binutils
../binutils-2.18/configure --target=arm-linux
make
make install
```

Danach befinden sich im Verzeichnis /usr/local die entsprechenden binutils Werkzeuge, sowohl im Unterverzeichnis /usr/local/bin mit der Zielplattform als Teil des Dateinamens (z. B. arm-linux-as), als auch im Verzeichnis /usr/local/arm-linux/bin ohne die Erweiterung des Dateinamens.

5.1.2.6 Konfiguration der Linux Kernel Header Quellen

Das Standardverzeichnis für den Linux Kernel-Quellcode ist /usr/src. Hier werden die Kernel-Quellen für eine konkrete Plattform und Kernelversion konfiguriert. Der Quellcode für dieses Beispiel wird entsprechend in das Verzeichnis /usr/src/linux-2.4.19-rmk6-pxa1-hh41.1 entpackt und dort konfiguriert. Die Konfiguration des Linux Kernels wird durchgeführt, bevor ein Linux Kernel übersetzt wird. In diesem Fall wurden die Quellen von www.handhelds.org bezogen und müssen entsprechend für die konkrete Zielplattform konfiguriert werden. Durch die Konfiguration werden C-Header Dateien für den für die Zielplattform konfigurierten Kernel erzeugt. Danach müssen die konfigurierten Header-Dateien in das Installationsverzeichnis der entstehenden Toolchain kopiert werden, so dass diese vom Übersetzer eingebunden werden können. Die Header-Dateien werden von der Toolchain benötigt, um die Aufrufe von Kernel-Funktionen in übersetzte Programme korrekt einbauen zu können. Diese Schritte müssen von einem Benutzer durchgeführt werden, der die entsprechenden Lese- und Schreibrechte besitzt.

```
cd /usr/src/linux-2.4.19-rmk6-pxa1-hh41.1
cp arch/arm/def-configs/ipaqpxa .config
make oldconfig
make include/linux/version.h
mkdir /usr/local/arm-linux/sys-root
mkdir /usr/local/arm-linux/sys-root/usr
mkdir /usr/local/arm-linux/sys-root/usr/include
```

```
cp -dR /usr/src/linux-2.4.19-rmk6-pxa1-hh41.1/include/asm-arm \
 /usr/local/arm-linux/sys-root/usr/include/asm
cp -dR /usr/src/linux-2.4.19-rmk6-pxa1-hh41.1/include/linux \
 /usr/local/arm-linux/sys-root/usr/include/linux
```

Die Header-Dateien sind nun für den Übersetzungsvorgang des GCC Compilers verfügbar.

5.1.2.7 Übersetzen des GCC Compilers ohne Sprachbibliothek

Der GCC Compiler kann ohne zusätzliche Komponenten nur Quellcode der Sprache C mit reduziertem Sprachumfang übersetzen. Um über mehr als den C-Sprachumfang zu verfügen, benötigt der GCC Compiler eine Sprachbibliothek, die an die Zielplattform angepasst ist und die Übersetzungsregeln für weitere Sprachen enthält. Diese muss aber mit einem Compiler übersetzt werden, der auf der Zielplattform ausführbaren Code erzeugt. Weder GCC Compiler noch Sprachbibliothek existieren zu diesem Zeitpunkt für die Zielplattform. Aus diesem Grund wird als nächstes ein rudimentärer GCC C-Cross-Compiler erzeugt, der anschliessend benutzt werden kann, um die angepasste Sprachbibliothek für die Zielplattform zu übersetzen.

Eine Datei aus den GCC Quellen muss für dieses Beispiel angepasst werden, um zu verhindern, dass beim Übersetzen Teile der Sprachbibliothek gesucht werden. In der Datei `gcc-3.4.6/gcc/config/arm/t-linux` muss folgende Zeile eingefügt werden:

```
T_CFLAGS = -Dinhibit_libc
```

Danach kann der GCC C-Compiler ohne Sprachbibliothek und Thread-Funktionalität übersetzt und installiert werden, wobei die Installation von einem Benutzer mit entsprechender Berechtigung für die Zielverzeichnisse durchgeführt werden muss:

```
cd gcc
../gcc-3.3.6/configure --target=arm-linux \
 --disable-threads --with-cpu=strongarm110 \
 --enable-languages=c --disable-shared \
 --with-sysroot
make
make install
```

Der erstellte Compiler befindet sich nun als `arm-linux-gcc` im Verzeichnis `/usr/local/bin` bzw. mit dem Namen `gcc` ohne Plattformpräfix im Verzeichnis `/usr/local/arm-linux/bin`. Die verwendeten Parameter haben folgende Bedeutungen:

target: Zielarchitektur, für die der Übersetzer ausführbaren Code erzeugen soll

disable-threads: Diese Übergangsversion des Übersetzers unterstützt noch keine Linux-Threads.

with-cpu: Genauere Spezifizierung der Prozessorarchitektur der Zielplattform

enable-language: Programmiersprachen, die von dem Übersetzer unterstützt werden sollen

disable-shared: Der erzeugte Übersetzer unterstützt keine Shared Libraries.

with-sysroot: Die für den Erstellungsvorgang benötigten Dateien der Zielplattform werden in einer Verzeichnisstruktur bereitgestellt, die einer Teilkopie der Wurzelverzeichnisses der Zielumgebung entsprechen. Wird kein Wert für diesen Parameter angegeben, erwartet der Erstellungsprozess diese Verzeichnisstruktur in einem Verzeichnis „sys-root“ unter dem Installationsverzeichnis des Compilers, in diesem Fall also /usr/local/arm-linux/sys-root. Dieses Verzeichnis entspricht dem Wurzelverzeichnis der Zielplattform.
Aus diesem Grund wurden auch die Kernel Header-Datein (die sich auf der Zielplattform im Verzeichnis `/usr/include` befinden) in das entsprechende Verzeichnis unter sys-root kopiert.

Wie bei den binutils werden für die Parameter –prefix, –build und –host sinnvolle Werte ermittelt bzw. Standardwerte angenommen.

Während des Erstellungsvorgangs wird der aktuelle Fortschritt laufend mitprotokolliert. Sollten beim Erstellen Fehler auftreten, ist es hilfreich, die entsprechenden Protokolldateien zu analysieren. Für die GCC werden zahlreiche Protokolldateien mit der Dateiendung .log erstellt, die in den diversen Unterverzeichnissen der verwendeten Komponenten zu finden sind.

5.1.2.8 Übersetzen des Sprachbibliothek glibc

Das Erstellen der Sprachbibliothek funktioniert analog zum Übersetzungsvorgang für die binutils und den GCC Compiler. Auch in den Quellen der Sprachbibliothek muss eine Korrektur vorgenommen werden, damit der Übersetzungsvorgang fehlerfrei durchläuft. In der Datei `glibc-2.3.6/Makeconfig` werden die beiden Zeilen

```
gnulib := -lgcc $(libgcc_eh)
static-gnulib := -lgcc -lgcc_eh $(libunwind)
```

wie folgt geändert:

```
gnulib := -lgcc
static-gnulib := -lgcc $(libunwind)
```

Auch hier muss die Installation, also der letzte Befehl, von einem Benutzer mit root Berechtigung ausgeführt werden.

```
cd glibc
../glibc-2.3.6/configure --prefix=/usr/local/arm-linux \
 --host=arm-linux --enable-add-ons=linuxthreads \
 --with-headers=/usr/local/arm-linux/sys-root/usr/include
make
make install
```

Das Konfigurationsskript ermittelt den benötigten Cross-Compiler und erstellt die Datei „Makefile“ derart, dass dieser für die Übersetzungsschritte verwendet wird.

5.1.2.9 Übersetzen des GCC Übersetzers mit Sprachbibliothek

Da die Sprachbibliothek für die Zielplattform nun zur Verfügung steht, muss der GCC Compiler erneut übersetzt werden, um die durch die Bibliothek angebotene Funktionalität nutzen zu können. Der bereits erstellte Compiler wird also in Kombination mit der Sprachbibliothek benutzt, um sich selbst neu zu erstellen, alledings mit erweitertem Funktions- und Sprachumfang.

Damit der Compiler die Sprachbibliothek glibc diesmal korrekt einbinden kann, muss zuerst die vorher in der Datei `gcc-3.4.6/gcc/config/arm/t-linux` eingefügte Erweiterung wieder rückgängig gemacht werden. Die Zeile

```
T_CFLAGS = -Dinhibit_libc
```

muss also wieder entfernt oder auskommentiert werden. Bevor der Compiler neu übersetzt wird, wird das Ergebnis des letzten Übersetzungsvorgangs des Compilers entfernt und dann erneut die Übersetzung und Installation durchgeführt.

```
cd gcc
rm -rf *
../gcc-3.4.6/configure --target=arm-linux \
 --with-cpu=strongarm110 --with-sysroot
make
make install
```

Jetzt steht der neue GCC Cross-Compiler im Verzeichnis `/usr/local/bin` als `arm-linux-gcc` bzw unter `/usr/local/arm-linux/bin` unter dem Namen `gcc` zur Verfügung.

5.1.3 GNU Project Debugger (GDB)

Der *GNU Project Debugger* (*GDB*) ermöglicht es, den Ablauf eines Programms zum Ausführungszeitpunkt zu verfolgen und zu untersuchen. Zu finden ist das

Projekt auf der Seite http://www.gnu.org/software/gdb/. Der zweite Aspekt ist die nachträgliche Analyse von Programmabstürzen.

Mit dem Parameter `-g` fügen die GCC beim Übersetzungsschrit zusätzliche Informationen in das Übersetzungsergebnis ein, die dann zur Laufzeit von einem Debugger wie GDB ausgewertet werden können.

Die für das Debugging benötigten Informationen im übersetzten Programm können im Nachhinein mit dem Befehl `strip` entfernt werden.

5.2 Hilfsmittel

Neben den zentralen Werkzeugen der unterschiedlichen Programmiersprachen, bietet Linux eine Vielzahl hilfreicher Programme, die die Entwicklungsarbeit erleichtern und auch in diesem Buch verwendet werden.

5.2.1 patch

Beim Erstellen von Software unter Linux kommt es immer wieder vor, dass für einen erfolgreichen Erstellungsvorgang die eine oder andere Quellcode- oder Konfigurationsdatei speziell angepasst werden muss. Um derartige Veränderungen zu vereinfachen, zu dokumentieren und kontrolliert und reproduzierbar durchzuführen, werden sie in Form so genannter *Patches* verteilt.

Eine Patch-Datei wird mit dem Werkzeug `diff` erstellt. `diff` ist Teil der GNU Diffutils (http://www.gnu.org/software/diffutils) und vergleicht zwei Dateien zeilenweise. Das Ergebnis ist die Liste der Unterschiede, die mit einer speziellen Syntax gekennzeichnet sind. Für ein einfaches Beispiel wird eine Datei `Datei1.txt` mit drei Zeilen angelegt.

```
Zeile 1
Zeile 2
Zeile 3
```

Die vorgesehene Änderung an der Datei ist das Löschen der zweiten Zeile. Dazu wird eine Kopie der Originaldatei mit dem Namen `Datei2.txt` angelegt und die Änderung vorgenommen. Das Ergebnis eines Vergleichs der beiden Dateien mittels `diff -Naur` führt zu folgendem Ergebnis.

```
--- Datei1.txt 2011-03-22 20:39:42.187472001 +0100
+++ Datei2.txt 2011-03-22 20:40:12.283472000 +0100
@@ -1,3 +1,2 @@
 Zeile 1
-Zeile 2
 Zeile 3
```

Dieser Inhalt wird in eine Patch-Datei umgeleitet. Um mit `diff` die Ausgabe im Patch-Format zu erzeugen, werden üblicherweise die Parameter `-Naur` verwendet. Damit können auch zwei Verzeichnisse verglichen und ein Patch erzeugt werden.

-N Fehlende Dateien werden behandelt, als ob es sich um leere Dateien handelt.
-a Alle Dateien werden als Textdateien verarbeitet.
-u An jeder Stelle, die verändert wird, werden 3 Zeilen des zusammengeführten Inhalts angezeigt.
-r Es werden rekursiv auch enthaltene Unterverzeichnisse verglichen.

```
diff -Naur Datei1.txt Datei2.txt > Datei.patch
```

Mit dem Werkzeug `patch` kann die Änderung, die an der ersten Datei vorgenommen werden soll, durchgeführt werden, indem die Patch-Datei ausgelesen und der Inhalt als Eingabe umgeleitet wird.

```
patch -p0 < Datei.patch
```

In diesem Fall müssen sich sowohl die zu verändernde Datei, als auch die Patch-Datei im aktuellen Arbeitsverzeichnis befinden. In der Patch-Datei werden die zu verändernden Dateien evtl. mit Pfadangabe referenziert. Befinden sich die Dateien in einer flacheren Verzeichnisstruktur, können mit dem Parameter `-p` führende Pfadbestandteile entfernt werden, damit die benötigten Dateien von `patch` gefunden werden. Mit `-p0` werden Dateinamen unverändert verarbeitet. Befindet sich beispielsweise in der Patch-Datei eine Dateireferenz `/home/cc/Datei1.txt`, kann mit dem Parameter `-p3` erreicht werden, dass die Datei unter `Datei1.txt` gesucht wird. Massgeblich ist der Schrägstrich als Verzeichnistrennzeichen. Entfernt werden die im Parameter `-p` angegebene Anzahl Schrägstriche sowie die Verzeichnisnamen davor.

Wenn `patch` nicht alle Veränderung eindeutig zuordnen kann, also Konflikte auftreten, generiert `patch` eine Datei mit der Dateiendung `.rej` oder `#`, in dem die problematischen Veränderungsstellen gespeichert werden.

5.2.2 make

Um die Abhängigkeiten zwischen den eingesetzten Werkzeugen, sowie zwischen den Teil- und Endergebnissen des Erstellungsprozesses zu verwalten und zu dokumentieren, wird sehr häufig das Werkzeug `make` eingesetzt. Dazu liest `make` die Datei `Makefile` aus und berechnet die Reihenfolge der auszuführenden Schritte aus den in der Makefile-Datei beschriebenen Abhängigkeiten. In der Datei `Makefile` werden die zu erstellenden Ergebnisse als so genannte *Ziele* (*engl.: Target*) definiert. Eine `Makefile`-Datei besteht also aus einem Satz an Regeln, aus welchen Eingabedateien mit welchen Werkzeugen welcher Ergebnisse erzeugt werden können. Ziele können auch Konfigurationszustände sein, wie die Installation der Erstellungsprodukte oder das Entfernen sämtlicher generierter Dateien.

`make` prüft bei der Verarbeitung einer `Makefile`-Datei, ob das jeweilige Ziel noch aktuell ist, oder neu erstellt werden muss. Dadurch kann die Gesamtdauer des

Erstellungslaufes reduziert werden, wenn nur Teile der zu erstellenden Software geändert wurden.

Ziele werden durch einen eindeutigen Namen gefolgt von einem Doppelpunkt definiert. Falls ein Ziel von anderen (Unter-)Zielen abhängt, die dementsprechend vorher erstellt werden müssen, werden die Namen der Ziele nach dem Doppelpunkt jeweils mit einem *Whitespace*-Zeichen getrennt. In den folgenden Zeilen werden die Befehle bzw. Programme aufgeführt, die aufgerufen werden müssen, um das jeweilige Ziel zu erstellen. Diese Zeilen müssen mit einem Tabulator-Zeichen beginnen. Kommentare beginnen mit einem # und reichen bis zum Zeilenende.

```
# Beispiel fuer eine Makefile-Datei mit den Zielen 'all'
# und 'ziel_1', wobei 'all' von 'ziel_1' abhaengt.
ziel_1:
befehl_1

all: ziel_1
befehl_2
```

Um häufig benutzte Werte an zentraler Stelle und nur einmal zu definieren, werden so genannte *Makros* eingesetzt. Makros beginnen mit einem Dollarzeichen (`$`), gefolgt von dem Makronamen. Variablen, die als Makronamen verwendet werden sollen, werden in runde Klammern gesetzt. Es gibt eine Reihe vordefinierter Makros, z. B. gibt `$` das aktuelle Ziel an.

Zieldefinitionen werden sowohl verwendet, um Abhängigkeiten zwischen den Zwischen- und Endprodukten festzulegen, als auch, um das Gesamtprojekt oder Teilbereiche gezielt zu erstellen. Um ein bestimmtes Ziel zu erstellen, wird `make` der entsprechende Zielname als Argument übergeben.

```
make ziel_1
```

5.2.3 Wget

Mit dem *GNU Wget* Paket, zu finden auf der Webseite http://www.gnu.org/software/wget, können Dateien über die Protokolle HTTP, HTTPS und FTP von Web- und FTP-Seiten heruntergeladen werden. Im Gegensatz zu interaktiven Web-Browsern funktioniert das Werkzeug `wget` über die Befehlszeile und kann daher gut in automatisierte Vorgänge eingebunden oder in einer Terminal-Sitzung verwendet werden. Z. B. kann die Dokumentation von Wget mit folgendem Befehl bezogen werden.

```
wget http://www.gnu.org/software/wget/manual/wget.html
--2011-03-23 19:17:45--  http://www.gnu.org/software/wget/manual/
wget.html
Aufloesen des Hostnamen www.gnu.org.... 140.186.70.148
Verbindungsaufbau zu www.gnu.org|140.186.70.148|:80... verbunden.
HTTP Anforderung gesendet, warte auf Antwort... 200 OK
Laenge: 291495 (285K) [text/html]
```

```
In wget.html speichern.

100%[======================================================================
==============>] 291.495     81,4K/s   in 3,5s

2011-03-23 19:17:49 (81,4 KB/s) - wget.html gespeichert [2914
95/291495]
```

Eine weitere hilfreiche Eigenschaft von Wget ist die Möglichkeit, eine ganze HTML-Hierarchie von der angegebenen Adresse zu beziehen. Damit lässt sich konfortabel eine lokale Kopie einer Webseite erstellen.

5.2.4 pkg-config

Das Werkzeug *pkg-config* unterstützt den Übersetzungsvorgang von Software, unabhängig von der Programmiersprache. Im Internet findet man das Programm unter pkg-config.freedesktop.org. pkg-config liest Metainformationen über ein Softwarepaket bzw. eine Bibliothek aus Dateien mit der Endung `.pc` aus und liefert die Informationen an das aufrufende Programm zurück.

Für diese Dateien gilt die Namenskonvention, dass an den Namen der Bibliothek die Dateiendung `.pc` angehängt wird. Gesucht wird in einem Verzeichnis `pkgconfig`, das sich im gleichen Verzeichnis wie die Bibliothek selbst befindet. Wenn z. B. die Bibliothek `bluez` im Verzeichnis `/usr/lib` installiert ist, erwartet pkg-config folgende Datei: `/usr/lib/pkgconfig/bluez.pc`. Die Liste der zu durchsuchenden Verzeichnisse kann über die Umgebungsvariable `PKG_CONFIG_PATH` um zusätzliche Verzeichnisse erweitert werden, die jeweils mit einem Komma getrennt sind. Die in `PKG_CONFIG_PATH` definierten Verzeichnisse werden immer vor den mittels Namenskonvention ermittelten Verzeichnissen durchsucht.

Die Syntax für Metadateien sieht Variablendefinitionen und Schlüsselwort-Wert-Paare vor. Variablen werden durch ein Gleichheitszeichen zugewiesen und über ein Dollarzeichen, gefolgt durch den Variablennamen in geschweiften Klammern in der Datei referenziert. Auf ein Schlüsselwort folgt ein Doppelpunkt und anschliessend der Wert.

```
home_dir=/home/cc
package_dir=${home_dir}/pkg

Name: Beispiel
URL: http://www.springer.de
```

In dem Beispiel wird zunächst die Variable `home_dir` definiert und deren Wert in der Zuweisung der Variable `package_dir` verwendet. Anschliessend werden

die Schlüsselworte `Name` und `URL` mit Werten belegt. Schlüsselworte werden von pkg-config vorgegeben, momentan gibt es folgende Definitionen:

Name:	Ein für Menschen lesbarer und verständlicher Name für das Paket.
Description:	Eine über den Namen hinausgehende Beschreibung des Pakets.
Version:	Die aktuelle Versionsnummer des Pakets.
URL:	Adresse der Webseite mit Informationen zu dem Paket.
Requires:	Eine durch Kommas getrennte Liste anderer Pakete, die dieses Paket benötigt. Falls notwendig kann mit den mathematischen Zeichen =, >, <, >= und <= eine bestimmte Versionsnummer angegeben werden.
Conflicts:	Mit diesem Schlüsselwort kann auf Unstimmigkeiten bzgl. der Versionen oder ganzer Pakete hingewiesen werden. Die Syntax entspricht der des Schlüsselworts `Requires:` mit dem Unterschied, dass ein Paket mehrfach aufgelistet werden kann, so dass sehr genau bestimmte Versionen eines Pakets ausgeschlossen werden können.
Libs:	Dieses Schlüsselwort definiert die Parameter für den Linker, um Programme mit diesem Paket zu verwenden.
Libs.private:	Diese Zeile definiert Bibliotheken, die der Linker für statische Bibliotheken benötigt.
Cflags:	Hiermit werden die Parameter definiert, die für einen Übersetzungsvorgang mit dieser Bibliothek benötigt werden.

Die wichtigsten Parameter für den Aufruf von pkg-config sind folgende:

–cflags	Dieser Parameter sorgt dafür, dass die Parameter zurückgegeben werden, die für einen Übersetzungsvorgang mit dem Inhalt des Pakets benötigt werden.
–libs	Mit dieser Option können die Parameter ermittelt werden, die der Linker benötigt, um Programme an dieses Paket zu binden.
–variable=NAME	Dieser Parameter gibt den Wert zurück, der in einer Metainformationsdatei für den gewünschten Parameter definiert wurde.
–define-variable=NAME=WERT	Damit kann eine Variablendefinition in einer `.pc`-Datei überschrieben werden, so dass der in der Datei definierte Wert nicht mehr gültig ist.

Normalerweise werden `.pc`-Dateien von dem Skript `configure` (Abschn. 5.2.6) erzeugt und nicht manuell erstellt.

Die Metadatei für bluez sieht beispielsweise folgendermassen aus:

```
prefix=/usr
exec_prefix=${prefix}
libdir=/usr/lib
includedir=${prefix}/include

Name: BlueZ
Description: Bluetooth protocol stack for Linux
Version: 3.18
Requires:
Libs: -L${libdir} -lbluetooth
Cflags: -I${includedir}
```

Somit können Programme, die Funktionalität dieser Bibliothek verwenden, mit folgender Befehlszeile übersetzt werden, wobei `quelltext.c` den Quelltext beinhaltet und `programmname` der Name des zu erstellenden Programms ist:

```
gcc quelltext.c -o programmname `pkg-config --cflags \
 --libs bluez`
```

Die rückwärtsgerichteten einfachen Anführungszeichen sorgen dafür, dass das Ergebnis des Aufrufs von `pkg-config` als Parameter an den Übersetzer in die Befehlszeile eingefügt wird.

5.2.5 Das Source Kommando

Mit Hilfe des so genannten *Source* Kommandos lassen sich Shell-Skripte innerhalb der Shell ausführen, aus der heraus sie aufgerufen werden. Im Gegensatz zum regulären Aufruf eines Shell-Skripts wird bei der Verwendung des Source Kommandos keine Unter-Shell erzeugt, in der das aufgerufene Skript ausgeführt wird. Normalerweise wird für ein Skript ein neuer Prozess gestartet, in diesem Prozess wird das Skript ausgeführt und nach dem Ausführen des Skripts wird der Prozess beendet.

Um nun ein Skript mit dem Source Kommando auszuführen wird dem Dateinamen des Skripts ein Punkt und ein Leerzeichen vorangestellt.

```
. script.sh
```

Auf diese Weise wird das Skript innerhalb der Shell ausgeführt, aus der heraus es aufgerufen wurde. Sämtliche Veränderungen, die das Skript in der Shell durchführt, bleiben nach dem Ende des Skripts in der Shell erhalten. Insbesondere die Werte von Umgebungsvariablen behalten Gültigkeit. Dieses Verhalten lässt sich sehr gut für die Konfiguration einer Entwicklungsumgebung einsetzen, in dem z. B. Pfade zu Werkzeugen wie Übersetzer, Linker usw. mittels Umgebungsvariablen gesetzt werden.

5.2.6 Die GNU Autotools und das GNU Build System

Das *GNU Buildsystem* definiert einen De-Facto-Standard, wie Programme und Pakete aus dem GNU-Projekt reproduzierbar und möglichst plattformunabhängig erstellt und die Erstellungsregeln gepflegt werden. Dieser Standard hat sich über das GNU-Projekt hinaus durchgesetzt, er vereinfacht die Erstellung und Portierung von Software, indem Unterschiede zwischen verschiedenen Plattformen in der Entwicklungsumgebung analysiert und in die Erstellungsvorschriften für das Projekt integriert werden.

Die *GNU Autotools* sind ein Programmpaket, das hauptsächlich aus den Werkzeugen *Autoconf*, *Automake* und *Libtool* besteht und den Entwickler beim Schreiben der benötigten Dateien weitgehend unterstützt, die für ein funktionierendes GNU Build System benötigt werden. Zu finden sind diese Werkzeuge auf ihren entsprechenden GNU Webseiten:

Autoconf: http://www.gnu.org/software/autoconf
Automake: http://www.gnu.org/software/automake
Libtool: http://www.gnu.org/software/libtool

Software, die als Quelltext ausgeliefert wird, muss, bevor sie einsatzbereit ist, zunächst meistens noch übersetzt (es sei denn, der Quelltext wird interpretiert), konfiguriert und installiert werden. In vielen Fällen, vor allem wenn es sich um C-Programme handelt, wird zur Steuerung des Erstellungsvorgangs das Werkzeug `make` eingesetzt. Dieses Programm liest die Erstellungsschritte aus einer Datei mit dem Namen `Makefile`, die mit dem Quellcode der Software ausgeliefert wird.

Der prinzipielle Ablauf bei der Verwendung der GNU Autotools sowie die verwendeten Dateien werden in Abb. 5.1 dargestellt und in diesem Kapitel beschrieben. Die Pflege von `Makefile` Dateien ist relativ aufwändig und fehleranfällig, ausserdem gibt es Unterschiede in der Syntax diverser Implementierungen von `make`. Daher wird oftmals stattdessen eine Skript-Datei mitgeliefert, die die Umgebung, in der die Erstellung stattfindet, analysiert und die benötigte `Makefile` Datei generiert. Das hat neben der vereinfachten Pflege der Erstellungsschritte den zusätzlichen Vorteil, dass Informationen über die Umgebung in die Erstellungsvorschriften einfliessen können. Für den Benutzer reduziert sich damit die Erstellung der Software auf folgende zwei Aufrufe, vorausgesetzt, die benötigten Werkzeuge sind in der Umgebung vorhanden:

```
configure
make
```

Wie in Abschn. 5.2.4 beschrieben, können Programmbibliotheken mit dem Werkzeug *pkg-config* verwaltet und für die Verwendung in eigenen Programmen konfiguriert werden. Die dafür benötigten `.pc`-Dateien sind ein weiteres wichtiges Ergebnis von `configure`-Skripts.

In den meisten Fällen ist es nicht ratsam, den Erstellungsvorgang direkt aus dem Verzeichnis heraus zu starten, in dem sich die Quellcode-Dateien befinden. Ein

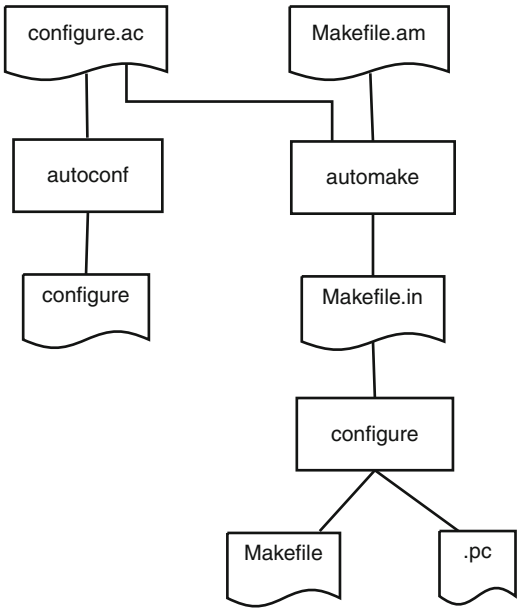

Abb. 5.1 Das GNU Buildsystem

gutes `configure`-Skript sollte immer den Aufruf aus einem anderen Verzeichnis heraus unterstützen.

5.2.6.1 GNU M4

Das Werkzeug Autoconf benötigt den Makroprozessor *GNU M4* für das Erzeugen der `configure`-Dateien. Dieses Programm verarbeitet Makros, indem es eine Eingabedatei einliest und die verwendeten Makros auswertet und ersetzt, um die Ausgabedatei zu schreiben. Informationen zu GNU M4 sind auf der Webseite http://www.gnu.org/software/m4 zu finden.

5.2.6.2 Autoconf

Autoconf erzeugt Shell-Skripte zur automatischen Konfiguration von Quellcode. Es besteht aus einer Sammlung erweiterbarer Makros für den Makroprozessor M4. Die produzierten Shell-Skripte hängen nicht von Autoconf oder M4 ab, zum Ausführungszeitpunkt, also in dem Moment, in dem ein Benutzer die Software installieren und verwenden will, wird weder Autoconf noch M4 benötigt. Ausserdem ist Perl (Abschn. 6.2.3) eine Voraussetzung für das Ausführen von Autoconf. Als Konvention hat sich `configure` als Name für die Skripte etabliert.

Wichtig für die Entwicklung von Software für das Pervasive Computing ist der Umstand, dass Autoconf Cross-Compiling unterstützt.

Eingabedateien für Autoconf heissen üblicherweise `configure.ac` oder in älteren Versionen `configure.in` und bestehen aus Makros, die die von dem zu konfigurierenden Quellcode deklarierten Voraussetzungen überprüfen. Ausserdem können die Dateien Anweisungen aus Shell-Skripten ausführen. Wird beim Aufruf von Autoconf kein Dateiname übergeben, liest das Programm die Datei `configure.ac` und schreibt das Ergebnis in die Datei `configure`. Andernfalls wird der Inhalt der übergebenen Datei ausgewertet und das Ergebnis auf die Standardausgabe gelenkt.

Die Konfigurationsskripte überprüfen auf der Plattform, auf der sie ausgeführt werden, das Vorhandensein diverser Funktionalität und geben entsprechend entweder passende Parameter für den Aufruf des dazugehörenden Werkzeugs oder Fehlermeldungen zurück. Auf diese Weise werden die Unterschiede diverser Plattformen überbrückt. Autoconf-Makros beginnen per Konvention mit der Zeichenfolge `AC_`. Beispielsweise kann mit dem Makro `AC_CHECK_HEADER` die Existenz bestimmter C-Headerdateien überprüft werden. Makros für Automake (Abschn. 5.2.6.3) werden über das Präfix `AM_` gekennzeichnet und Libtool-Makros beginnen mit `LT_`.

Autoconf liest die zu verarbeitende Datei ein, bis es auf ein Makro stösst. Der bis zu dem Makro gelesene Inhalt wird unverändert ausgegeben und findet sich somit in identischer Form in der generierten Datei. Falls das Makro Parameter benötigt, werden diese zuerst ausgewertet und ggf. darin enthaltene Makros rekursiv verarbeitet. Ist in den Parameterwerten Text in den Quotierungszeichen `[` und `]` eingeschlossen, wird dieser Text ohne die Quotierungszeichen unverändert ausgegeben und nicht weiter nach Makros durchsucht. Das Ergebnis einer Makroersetzung wird anstelle des Makros in den zu verarbeitenden Dateiinhalt eingefügt und die Verarbeitung an dieser Stelle, also vor dem ersetzten Text, fortgesetzt. Wenn das Ergebnis einer Makroauflösung wiederum Makros enthält, werden diese bei der weiteren Verarbeitung erkannt und entsprechend verarbeitet. Als Trennungszeichen für Parameter eines Makros dient das Komma. Das hat zur Folge, dass auszugebender Text, der Kommas oder Makros enthält, immer in Quotierungszeichen eingeschlossen werden muss.

Autoconf-Dateien müssen mit dem Initialisierungsmakro `AC_INIT` beginnen. Hier werden der Paketname und die Version als Parameter mitgegeben. Als als letztes Makro sollte sinnvollerweise `AC_OUTPUT` stehen, das dafür sorgt, dass das Ergebnis auch tatsächlich ausgegeben wird. Zwischen diesen beiden Makros werden die Makros für die benötigten Konfigurationsaufrufe und Funktionalitätstests eingefügt.

Die folgende `configure.ac`-Datei beispielsweise überprüft nach der Initialisierung die Existenz der Header-Dateien `stdio.h` und `stdlib.h` und erzeugt im Erfolgsfall die C-Präprozessordirektiven `#define HAVE_STDIO_H 1` und `#define HAVE_STDLIB_H 1`.

```
AC_INIT([verbrauch], [1.0])
AM_INIT_AUTOMAKE([foreign -Wall -Werror])
AC_CHECK_HEADERS([stdio.h])
AC_CHECK_HEADERS([stdlib.h])
```

```
AC_PROG_CC
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Die Zeile `AM_INIT_AUTOMAKE` initialisiert die Umgebung für Automake und stellt sicher, dass die von Automake benötigten Werkzeuge installiert sind. Der Konfigurationsparameter `foreign` sorgt dafür, dass die Einhaltung der GNU Kodierrichtlinien nicht zu streng geprüft wird. Mit `-Wall` werden alle auftretenden Warnungen angezeigt und mit `-Werror` als Fehler ausgegeben. Mit `AC_PROG_CC` wird sichergestellt, dass ein C-Übersetzer vorhanden ist. Das Makro `AC_CONFIG_FILES` sorgt dafür, dass die aufgelisteten Konfigurationsdateien erzeugt werden, und zwar als Eingabedateien für die passenden Autotools Werkzeuge. Im obigen Fall werden zwei `Makefile.in` Dateien ausgegeben.

5.2.6.3 Automake

Autoconf erwartet in Verzeichnissen, die Quellcode oder andere Dateien enthalten, die für die Erstellung und Installation des zu erstellenden Programmpakets notwendig sind, jeweils eine Datei `Makefile.in`. Das Werkzeug *Automake* unterstützt die Erstellung und Pflege von `Makefile.in`-Dateien. Der Preis dafür ist, dass Automake davon ausgeht, dass es in Zusammenhang mit Autoconf aufgerufen wird und die Ergebnisse von Autoconf eingelesen und verarbeitet werden. Automake liest auch die Datei `configure.ac` des Projektes aus und überprüft die Existenz benötigter Makros und Variablen. Somit besteht eine Abhängigkeit zu Autoconf.

Das GNU Buildsystem sieht als Konvention eine `Makefile`-Datei pro Unterverzeichnis vor. Automake produziert dafür aus den `Makefile.am`-Dateien die entsprechenden Steuerdateien, die die dazu benötigten Metainformationen für Autoconf enthalten.

Die Syntax von Automake-Dateien entspricht derjenigen von Makefile-Dateien, wobei Unterschiede in den zu erstellenden Zielprodukten über die Definition von Variablen gesteuert werden. Die Namen von Automake-Variablen haben eine semantische Bedeutung, aus der Automake Informationen zu Dateitypen und Zieldefinitionen extrahiert. Beispielsweise definieren Variablen mit dem Suffix `_PROGRAMS` die Namen der ausführbaren Programme, die mit der zu erstellenden Makefile-Datei gebaut werden sollen.

Per Konvention befindet sich der Quellcode eines Projektes für ausführbare Programme im Unterverzeichnis `src`. Um im Wurzelverzeichnis eines Projektes mit dem Erstellungsvorgang starten zu können, muss Automake angewiesen werden, im Verzeichnis `src` mit der Verarbeitung fortzufahren. Die initiale Datei `Makefile.am` ist daher sehr kurz.

```
SUBDIRS = src
```

In diesem Verzeichnis befindet sich dann die nächste `Makefile.am` Datei mit den ersten konkreten Erstellungsinformationen.

```
bin_PROGRAMS = verbrauch
verbrauch_SOURCES = verbrauch.c
```

Die erste Zeile sagt aus, dass ein ausführbares Programm mit dem Namen `verbrauch` erstellt und im Verzeichnis `bin` installiert werden soll. Mit der zweiten Zeile wird der Quellcode für das Programm `verbrauch` bekanntgemacht. Mehrere Dateien in der selben Zeile werden durch Leerzeichen getrennt.

Das Werkzeug `autoreconf` aktualisiert die Steuerdateien für die Autotools, indem es die entsprechenden Werkzeuge aufruft.

```
autoreconf --install
```

Der Parameter `--install` veranlasst `autoreconf` fehlende Dateien zu erstellen.

5.2.6.4 Libtool

Das plattformunabhängige Erstellen von Programmbibliotheken ist mindestens so komplex wie die Vorgehensweise bei ausführbaren Programmen. Libtool unterstützt bei der Erstellung sowohl von statischen als auch von gemeinsam genutzten Programmbibliotheken. Zunächst führt Libtool eine Abstraktionsschicht für das Format von Programmbibliotheken ein, das so genannte *Libtool Archive* mit der Dateiendung `.la`. In den Steuerdateien für Automake wird dieses Format verwendet, wenn eine Bibliothek eingebunden werden soll.

Die grundsätzliche Vorgehensweise ist ähnlich der für ausführbare Programme. In der Datei `configure.ac` im Wurzelverzeichnis des Projektes kommen im Vergleich zum Beispiel in Abschn. 5.2.6.2 drei Zeilen hinzu. Da für dieses Beispiel keine Standard-Header-Dateien benötigt werden, entfallen die Verfügbarkeitsprüfungen mit `AC_CHECK_HEADERS`.

```
AC_INIT([durchschnitt], [1.0])
AM_INIT_AUTOMAKE([foreign -Wall -Werror])
LT_INIT
AC_CONFIG_MACRO_DIR([m4])
AC_PROG_LIBTOOL
AC_PROG_CC
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([Makefile lib/Makefile])
AC_OUTPUT
```

Mit dem Makro `LT_INIT` wird die Umgebung für Libtool initialisiert und mit `AC_CONFIG_MACRO_DIR([m4])` ein Verzeichnis `m4` spezifiziert, das Libtool für benötigte Autoconf Makros verwenden soll. Das Makro `AC_PROG_LIBTOOL` stellt sicher, dass Libtool auf dem System installiert und verfügbar ist.

Der Quellcode für die Bibliothek befindet sich hier im Verzeichnis `lib`, daher verweist `AC_CONFIG_FILES` auf die `Makefile` Datei in diesem Unterverzeichnis.

Die Datei `Makefile.am` im Wurzelverzeichnis verweist ebenfalls auf das Unterverzeichnis `lib` und macht das Makroverzeichnis `m4` bekannt.

```
SUBDIRS = src
ACLOCAL_AMFLAGS = -I m4
```

In der Datei `Makefile.am` im Unterverzeichnis `lib` schliesslich wird definiert, dass eine Programmbibliothek erstellt werden soll.

```
lib_LTLIBRARIES = libdurchschnitt.la
libdurchschnitt_la_SOURCES = durchschnitt.c durchschnitt.h
```

Das Schlüsselwort `LTLIBRARIES` gibt an, dass die Bibliothek nach den Regeln und zur Verarbeitung von Libtool erzeugt werden soll, also als *Libtool Archive*. Der Präfix `lib` wiederum legt fest, dass das Ergebnis in das Verzeichnis `lib` installiert wird. Der Name der Bibliothek im Libtool-Format ist `libdurchschnitt.la` und dient als Präfix für die Angabe der zur Erstellung benötigten Dateien in der zweiten Zeile.

Nachdem das Verzeichnis `m4` für die Autoconf-Makros erstellt wurde, kann das Projekt mittels `autoreconf` initialisiert werden.

```
mkdir m4
autoreconf --install
```

Nun kann das Projekt mit `./configure` konfiguriert und mit `make` erstellt werden. Das Ergebnis im Unterverzeichnis `lib` ist neben der übersetzten Objektdatei `durchschnitt.o` die Programmbibliothek `libdurchschnitt.la`. Ausreichende Zugriffsrechte vorausgesetzt installiert `make install` die Bibliothek im Format der Zielplattform in das dafür vorgesehene Verzeichnis.

5.2.6.5 Besonderheiten bei Pervasive Linux

Abhängig davon, welche Schritte des GNU Build Systems auf dem Weg zur Zielplattform auf welchem System ausgeführt werden, ergeben sich Unterschiede im Ablauf und den Aufrufparametern.

Das Übersetzen des Quellcodes für das Ausführen auf einer anderen Plattform (Cross-Compilation) wird über den Paramter `--host` gesteuert, beispielsweise für eine ARM-Plattform.

```
./configure --host=arm-angstrom-linux-gnueabi
make
```

Der Übersetzungsvorgang findet in diesem Fall auf der Plattform statt, auf der das Projekt konfiguriert wurde.

Soll die Konfiguration für eine Erstellung auf einer anderen Plattform (üblicherweise die Zielplattform) erfolgen, kann dies mit dem Parameter `--build` entsprechend angegeben werden. Das konfigurierte Projekt muss dann auf die spezifizierte Plattform kopiert und dort erstellt werden.

5.2.7 qmake

Das Werkzeug *qmake* wird von der Firma Qt Software im Rahmen der Qt Bibliothek angeboten. qmake vereinfacht das plattformunabhängige Erzeugen von `Makefile` Dateien zur Steuerung des Übersetzungs- und Erstellungsprozesses von Anwendungsprojekten.

Das Programm kann unabhängig von der Qt Bibliothek verwendet werden, bietet jedoch für die Entwicklung von Software auf Basis von Qt besondere Unterstützung an.

Die Beschreibung der Erstellungsregeln für eine Makefile-Datei erfolgt in Projektdateien mit der Endung `.pro`. Dabei handelt es sich um einfache Textdateien. Falls in der Datei keine Regel für den Namen des Ergebnisses des Erstellungsprozesses spezifiziert wird, bestimmt ihn der Dateiname der `.pro`-Datei. In dieser Datei wird der Quellcode, ggf. der Name des Ergebnisses, sowie Plattformunterschiede konfiguriert und erforderliche Voraussetzungen und Anforderungen geprüft.

5.2.7.1 Variablen

Variablen bestehen aus Listen von Texteinheiten und können auf zweierlei Weise definiert werden. Entweder werden zusätzliche Einträge über den += Operator angehängt, oder jeder Eintrag in einer neuen Zeile mit dem Trennzeichen \ getrennt. Da das Leerzeichen als Trennzeichen zwischen zwei Werten eingesetzt wird, müssen Texteinheiten, die selbst ein Leerzeichen beinhalten in Anführungszeichen gesetzt werden.

```
SOURCES += durchschnitt.c
SOURCES += verbrauch.c
HEADERS = durchschnitt.h \
   berechne.h

DIR = "/home/cc/Qt Programme"
```

Neben `SOURCES` und `HEADERS`, mittels derer Quellen und Header-Dateien angegeben werden, sind folgende Variablen wichtig:

TARGET	legt den Namen des Erstellungsergebnisses fest. Falls die Variable fehlt, wird der Dateiname der `.pro` Datei verwendet.
CONFIG	definiert allgemeine Projektkonfigurationsoptionen.
FORMS	listet die Beschreibungsdateien für die Benutzerschnittstelle auf, die üblicherweise mit einem entsprechenden Werkzeug entworfen wurde.
TEMPLATE	legt den Typ des Ergebnisses fest, also ob eine Anwendung, eine Bibliothek oder ein Plug-In erstellt werden soll. Falls der Wert fehlt, wird eine `Makefile` Datei erstellt, die ein Anwendungsprogramm erzeugt.
DESTDIR	legt das Ergebnis in dem angegebenen Verzeichnis ab.

RESOURCES	erlaubt es, über das so genannte *Qt Resource System* Binärdaten wie z. B. Bilddateien in das Projekt zu integrieren.
QT	definiert Qt-spezifische Konfigurationsparameter.

Auf die Werte der so definierten Variablen kann mit einer an Linux-Systemvariablen angelehnten Syntax zugegriffen werden. Um den Wert einer Variablen auszugeben, werden vor den Variablennamen zwei Dollarzeichen gesetzt.

```
VARIABLE1 = $$VARIABLE2
```

5.2.7.2 Kommentare

Kommentare werden durch das Zeichen # eingeleitet und setzen sich bis zum Zeilenende fort.

```
# Kommentare erleichtern die Lesbarkeit von Quelltext.
```

5.2.7.3 Ablaufsteuerung

qmake stellt eine Reihe an Funktionen zur Verfügung mit Hilfe derer der logische Ablauf gesteuert werden kann. Z.B. kann über die Funktion `include` eine andere Projektdatei an der angegebenen Stelle eingefügt werden.

```
include(durchschnitt.pro)
```

Um schliesslich eine `Makefile` Datei für das Projekt zu erzeugen, wird qmake mit dem Parameter `-o` aufgerufen.

```
qmake -o Makefile verbrauch.pro
```

5.2.8 Ant

Aus dem Umfeld für die Entwicklung von Programmen für die Java Plattform ist das Erstellungswerkzeug *Apache Ant* (http://ant.apache.org) hervorgegangen. Ant ist selbst in der Programmiersprache Java geschrieben, lässt sich aber für den Erstellungsvorgang beliebiger Programmiersprachen universell einsetzen. Das Layout von Ant-Dateien ist weniger strikt, als beispielsweise `Makefile` Dateien, und daher weniger fehleranfällig bzgl. Formatierungen.

Die Erstellungsvorschriften für Ant werden mit XML (Abschn. 3.5.2.1) beschrieben und die auszuführenden Schritte als Java-Klassen implementiert. Ant selbst bringt bereits viele Implementierungen solcher Erstellungsanweisungen mit. Ant-Steuerdateien sind plattformunabhängig und können auf jedem System, für das Ant verfügbar ist, ausgeführt werden.

Jedes zu erstellende Projekt benötigt eine eigene Ant Datei, die wiederum mindestens ein zu erstellendes Ziel definiert. In den Zieldefinitionen werden die Schritte festgelegt, die zum Erstellen des Ziels nötig sind.

Wenn mit dem Parameter `-buildfile` keine andere Erstellungsdatei angegeben wird, erwartet Ant die Erstellungsvorschriften in einer Datei mit dem Namen `build.xml`. Die folgende Datei enthält die Erstellungsregeln für das Beispielprogramm aus Abschn. 6.2.2.3.

```
<project name="Verbrauch" default="programm" basedir=".">
    <description>
        Ant-Datei zum Erstellen des Programms "Verbrauch"
        zur Berechnung des durchschnittlichen Kraftstoff-
        verbrauchs eines Fahrzeugs.
    </description>

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="ergebnis"  location="ergebnis"/>

  <target name="init">
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="programm" depends="compile">
    <mkdir dir="${ergebnis}/lib"/>
    <jar jarfile="${ergebnis}/lib/verbrauch.jar"
         basedir="${build}"/>
  </target>
</project>
```

Die Datei beginnt mit dem Element `project`. Die optionalen Attribute `name`, `default` und `basedir` geben den Projektnamen an, definieren das Standardziel, das erstellt werden soll, wenn kein spezielles Ziel beim Aufruf von Ant mitgegeben wurde, und das Verzeichnis, von dem aus die relativen Pfadangaben in der Datei berechnet werden. Mit dem Element `description` können viele Elemente um hilfreiche Beschreibungen angereichert werden.

Mit den `property` Elementen werden Variablen definiert, die im weiteren Verlauf der Datei verwendet werden. Für den Quellcode ist in diesem Fall das Unterverzeichnis `src` vorgesehen, die Übersetzungsergebnisse werden im Verzeichnis `build` abgelegt und für die Installation vorgesehene Ergebnisse sollen im Verzeichnis `ergebnis` angelegt werden. Dafür wurden die gleichnamigen Variablen definiert.

Die erste Zieldefinition hat den Namen `init` und legt mit Anweisung `mkdir` das Unterverzeichnis für die Übersetzungsergebnisse an.

Der Übersetzungsvorgang wird über das Ziel `compile` gesteuert und enthält in diesem Fall auch nur eine einzige Anweisung. Über das Attribut `depends` wird signalisiert, dass das Ziel `init` vor dem Ziel `compile` ausgeführt werden muss. Auf diese Weise werden die Abhängigkeiten unter den Zieldefinitionen sichtbar.

Das letzte Ziel ist das oben definierte Standardziel `programm`. Es hängt von einem erfolgreichen Übersetzungsvorgang ab, erstellt das Verzeichnis `lib` im definierten Ergebnisverzeichnis und packt das übersetzte Programm für die Verteilung und Installation in die Java Archivdatei `verbrauch.jar` in dem zuvor erstellten Verzeichnis.

Der Aufruf `ant` liest die Datei `build.xml` ein, wertet die definierten Abhängigkeiten aus und erstellt die benötigten Ziele in der berechneten Reihenfolge. Soll beispielsweise der Quellcode nur übersetzt werden, wird Ant der entsprechende Zielname mit `ant compile` übergeben.

5.2.9 Maven

Das Apache-Projekt *Maven* wurde ursprünglich zur Vereinfachung des Erstellungsprozesses des Projektes *Jakarta Turbine* entwickelt. Durch die Weiterentwicklung entstand ein generelles Werkzeug zur Steuerung des Erstellungsvorgangs von Java-Projekten. Die Projektseite im Internet ist http://maven.apache.org. Maven ist selbst in Java geschrieben.

Das Programm arbeitet mit einem so genannten *Project Object Model* (*POM*), in dem Meta-Informationen zu einem Projekt abgelegt sind. Das POM wird als XML-Datei `pom.xml` im Wurzelverzeichnis des Projekts gespeichert.

Ein zentrales Konzept bei Maven ist der *Lebenszyklus des Erstellungsprozesses* (*engl.: Build Life Cycle*). Jeder Lebenszyklus besteht aus genau definierten *Phasen* (*engl.: Phases*), die ihrerseits *Ziele* (*engl.: Goals*) beinhalten, die sequenziell ausgeführt werden. Ziele können in mehreren Phasen vorkommen. Drei Lebenszyklen unterschiedlicher Erstellungsprozesse bringt Maven bereits mit. *Default* beinhaltet den Erstellungsprozess bis hin zur Installation, *Clean* das Löschen erstellter Dateien und *Site* die Generierung der Projektdokumentation. Zusammengehörende Ziele werden logisch in Plug-Ins gruppiert. Um Zieldefinitionen referenzieren zu können, wird der Name des Plug-Ins von dem des Ziels mit einem Doppelpunkt getrennt.

Ein einfaches Entwicklungsprojekt kann mit dem Befehl `mvn` angelegt werden.

```
mvn archetype:generate
 -DgroupId=com.springer.durchschnitt \
 -DarchetypeArtifactId=maven-archetype-quickstart \
 -DartifactId=Verbrauch -DinteractiveMode=false
```

Falls Maven bei der Ausführung zusätzliche Elemente benötigt, z. B. weil bestimmte Plug-Ins benötigt werden, werden diese aus dem Internet heruntergeladen und installiert.

Das Ziel `generate` im Plug-In `archetype` sorgt dafür, dass der Projektrahmen erzeugt wird, in diesem Fall ohne weitere Fragen an den Benutzer, was mit dem Parameter `-DinteractiveMode=false` signalisiert wird.

Damit wird das Wurzelverzeichnis mit dem Projektnamen angelegt, der mit dem Parameter `-DartifactId` übergeben wurde. In gleichnamigen Verzeichnis befindet sich die Datei `pom.xml`. Diese zentrale Konfigurationsdatei enthält u.a. die Parameterwerte für den Gruppennamen `groupId` und den eindeutigen Namen des zu erstellenden Ergebnisses `artifactId`. Die Wert für `groupId` entspricht dem zentralen Java-Paketnamen der zu entwickelnden Anwendung. Hier werden auch z. B. auch die Versionsnummer des Erstellungsergebnisses und die Art der Paketierung für die Verteilung festgelegt.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 \
                      http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.springer.durchschnitt</groupId>
  <artifactId>Verbrauch</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Verbrauch</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Unterhalb des Wurzelverzeichnisses befindet sich das Unterverzeichnis `src`, das den Quellcode des Projekts aufnehmen soll. Dabei unterscheidet Maven von vornherein zwischen dem produktiven Anwendungsquellcode im Unterverzeichnis `main` und einem dazu parallelen Verzeichnisbaum, in dem automatisierte *Unit Tests* auf Basis des bei der Entwicklung von Java-Anwendungen weit verbreiteten Test-Rahmenwerks *JUnit* entstehen sollen. Das ist die Verzeichnisstruktur, die Maven als Standard vorsieht. Wird diese Konvention eingehalten, benötigt Maven keine zusätzlichen Informationen bzgl. des Speicherorts von Quellcode.

Mit dem Ziel `generate` können unterschiedliche Projektgerüste generiert werden. Im Beispiel wurde `maven-archetype-quickstart` als Wert für den Parameter `-DarchetypeArtifactId` gewählt. Dieser Projekttyp erzeugt eine einfache Java-Klasse zur Veranschaulichung des Funktionsprinzips, die gelöscht werden kann, sobald mit der Eigenentwicklung begonnen wird. In diesem Beispiel muss der Quellcode aus Abschn. 6.2.2.3 in das Verzeichnis `src/main` kopiert werden.

So wie Erstellungsziele können auch Phasen direkt beim Aufruf von Maven angegeben werden. In diesem Fall werden die Phasen des Lebenszyklus bis zu der genannten ausgeführt und der Erstellungsprozess nach dieser abgebrochen.

```
mvn package
```

Dieser Aufruf führt den Default Lebenszyklus bis zur Phase `package` aus. Dies schliesst die Phasen `validate`, `compile`, `test` und `package` ein. Das Ergebnis ist in diesem Fall die Datei `1.0-SNAPSHOT.jar` im Unterverzeichnis `target` des Projekts.

Maven verwaltet im Benutzerverzeichnis eine Ablage für Projektergebnisse, mit Hilfe derer auf Programmcode anderer Projekte zugegriffen werden kann. Diese Ablage wird in einem Unterverzeichnis `.m2` angelegt.

Für die Integration in die Entwicklungsumgebung Eclipse (Abschn. 5.4) sorgt das Projekt M2Eclipse auf der Webseite http://m2eclipse.sonatype.org.

5.2.10 BitBake

Das in der Skriptsprache Python (Abschn. 6.2.4) geschriebene Steuerwerkzeug *BitBake* steuert die Einzelschritte (engl.: Tasks) eines Erstellungsvorgangs für Softwarepakete und verwaltet die dazugehörenden Metadaten. Zu finden ist BitBake unter http://developer.berlios.de/projects/bitbake. Die Grundkonzepte wurden von *Portage* übernommen, mit Hilfe dessen die Linuxdistribution *Gentoo* den Erstellungsprozess steuert. Dass BitBake in Python geschrieben ist, hat den Vorteil, dass die Skipte nicht übersetzt werden müssen und daher sehr plattformunabhängig sind.

5.2.10.1 Installation

Für die Installation kann BitBake in ein beliebiges Verzeichnis entpackt werden, das über die in der Umgebungsvariable `PATH` definierte Verzeichnisliste erreichbar ist oder gemacht wird.

```
cd /home/cc
gunzip bitbake-1.10.1.tar.gz
tar -xvf bitbake-1.10.1.tar
export PATH=$PATH:/home/cc/bitbake-1.10.1/bin
```

5.2.10.2 Dateitypen

BitBake arbeitet mit drei Dateitypen: Konfigurationsdateien, Klassen und Steuerdateien.

Dabei erwartet BitBake die zentrale Konfigurationsdatei `bitbake.conf` in einem fest vorgegebenen Verzeichnis `conf`, das sich in der Liste der Verzeichnisse befindet, die in der Umgebungsvariable `BBPATH` spezifiziert ist. In dieser Datei werden die für ein Projekt gültigen Umgebungsvariablen gesetzt und möglicherweise weitere Konfigurationsdateien referenziert. Eine Beispieldatei `bitbake.conf`, die als Referenz möglicher Variablendefinitionen bzw. als Schablone für die eigene Entwicklung verwendet werden kann, wird in der Archivdatei von BitBake mitgeliefert.

Für den Erstellungsablauf nutzt BitBake Steuerdateien, die die Endung `.bb` haben und Klassen, die in Dateien mit der Dateiendung `.bbclass` gespeichert werden. Die Steuerdateien werden auch *Rezepte* (*engl.: Receipes*) genannt. Über die Klassen wird ein rudimentäres Vererbungs- bzw. Bibliothekskonzept realisiert. Diese Dateien erwartet BibBake in einem Verzeichnis mit dem Namen `classes` relativ zu einem in der Umgebungsvariable `BBPATH` angegebenen Pfad.

Die eigentliche Abfolge der Erstellungsschritte wird in den `.bb` Dateien gesteuert. Üblicherweise existiert eine `.bb` Datei pro zu erstellendem Paket, wobei Abhängigkeiten zwischen den zu erstellenden Paketen von BitBake beachtet werden.

5.2.10.3 Syntax

Die Syntax der Steuerdateien von BitBake ist eine Mischung aus Variablendeklarationen, wie sie in Shellskripten verwendet werden und Funktionsaufrufen der Skriptsprache Python.

Variablen werden über den Zuweisungsoperator = definiert und mit der Syntax `${variablenname}` referenziert. Im folgenden Beispiel wird eine Variable `verzeichnis` definiert und bei der Definition der zweiten Variable `pfad` verwendet.

```
verzeichnis = "/home/cc"
pfad = ${verzeichnis}
```

Über einen Satz an vordefinierten Variablen, kann bzw. muss BitBake bestimmte Meta-Information über das in der Steuerdatei beschriebene Paket bekannt gemacht werden. Beispielsweise kann mit der Variablen `DESCRIPTION` eine kurze Beschreibung des Pakets vorgenommen und mit `AUTHOR` der Author benannt werden.

Die zur Erstellung eines Projektes benötigten Dateien kann BitBake von unterschiedlichen Quellen beziehen. Mögliche Quellen sind neben dem lokalen Dateisystem z. B. CVS-, HTTP-, FTP-, SVN- oder Git-Server. Die eigentliche Datei wird dabei über eine URI definiert, wobei der Protokolltyp den zu verwendenden Lademechanismus beschreibt, ähnlich wie das in einem Web-Browser der Fall ist. Eine Datei im lokalen Dateisystem wird beispielsweise mit dem Protokoll `file://` beschrieben.

```
SRC_URI= "file:///home/cc/springer/durchschnitt.h"
```

Erstellungsschritte werden als eine Art Funktionsbeschreibung definiert, deren Inhalt von BitBake ausgeführt wird. Die Namen dieser Funktionsbeschreibungen müssen mit `do_` beginnen und werden mit dem Aufruf `addtask` deklariert. Das Schlüsselwort `python` vor der Funktionsdeklaration teilt BitBake mit, dass es sich um ausführbaren Python-Code handelt. In der `addtask`-Zeile wird der neue Erstellungsschritt ohne den Präfix `do_` angeführt. Zusätzlich können Abhängigkeiten zu anderen Erstellungsschritten spezifiziert werden. Dies funktioniert über einen Angebot/Nachfrage-Mechanismus. Pakete, die eine bestimmte Funktionalität nach aussen, also für andere Pakete anbieten, signalisieren das durch die Erweiterung des

Wertes der Variablen PROVIDES. Umgekehrt deklarieren Pakete ihre Abhängigkeit von dem Vorhandensein anderer Paktete durch das Schlüsselwort DEPENDS.

5.2.10.4 Aufruf

Der Befehl bitbake startet den Erstellungsvorgang. Die zu verarbeitenden Steuerdateien erwartet BitBake entweder über die Definition der Umgebungsvariablen BBFILES oder als Wert für den Übergabeparameter -b. Über die Umgebungsvariable BBFILES können auch mehrere durch Leerzeichen getrennte Steuerdateien übergeben und die Dateinamen maskiert, also mit Platzhaltersymbolen (engl.: Wildcards) versehen werden. Der Parameter -b akzeptiert nur eine einzelne Steuerdatei. Üblicherweise entspricht der erste Teil des Dateinamens dem in der Datei beschriebenen Paketnamen.

Der auszuführende Erstellungsschritt bzw. Task wird mit dem Übergabeparameter -c angegeben. Falls dieser Parameter fehlt, nimmt BitBake build als den auszuführenden Schritt an.

Wenn BitBake über die Umgebungsvariable BBFILES aufgerufen wird, benötigt BitBake noch den Paketnamen, für den der Erstellungsschritt ausgeführt werden soll, denn es kommt vor, dass die Steuerdateien unterschiedlicher Pakete, die selben Bezeichnungen für Erstellungsschritte definieren. Das ist beim Aufruf mit dem Parameter -b nicht nötig, da die Namen der Erstellungsschritte innerhalb einer Steuerdatei eindeutig sein müssen. Aus dem angegebenen Erstellungsschritt und der daraus resultierenden Abhängigkeiten erstellt BitBake eine Liste von Erstellungsschritten und führt diese der Reihe nach aus.

Für die Erstellung eines Softwarepakets mit BitBake müssen mindestens die Steuerdateien (.bb-Dateien) beschafft werden und optionalerweise auch diejenigen Quelldateien, auf die im lokalen Dateisystem zugegriffen werden soll. Wenn die für die Erstellungsschritte benötigten Quelldateien für BitBake zugreifbar sind, also z. B. direkt aus dem Netz geladen werden oder lokal verfügbar gemacht wurden, reicht es aus, die zu dem Paket gehörenden Steuerdateien mit Hilfe der Umgebungsvariablen BBFILES aufzulisten, um den Erstellungsvorgang starten zu können.

5.2.11 crosstool

Die zeitaufwändigen und teilweise zermürbenden Schritte für die Erstellung einer Toolchain für Cross Compilation veranlassten Dan Kegel den Prozess mit einem ausgefeilten, plattformunabhängigen Skript zu automatisieren. Das Ergebnis ist *crosstool* und kann von der Webseite http://kegel.com/crosstool heruntergeladen werden. Dieses äusserst hilfreiche Werkzeug übernimmt sämtliche Schritte zur Erzeugung einer auf der Sprachbibliothek glibc basierenden Toolchain vom Herunterladen aller benötigten Quellen, über deren Modifikation und Konfiguration, Einspielen benötigter Patches bis zum Übersetzungsvorgang und anschliessender Installation.

Zunächst werden die crosstool Quellen entpackt:

```
gunzip crosstool-0.42.tar.gz
tar -xvf crosstool-0.42.tar
cd crosstool-0.42
```

Im Verzeichnis `crosstool-0.42` befinden sich u.a. eine Reihe beispielhafter Skriptdateien, für jede unterstützte Plattform eine. Die Dateinamen beginnen mit `demo-` gefolgt von der entsprechenden Plattform und haben die Dateiendung `.sh`, also z. B. `demo-ppc405.sh` für einen PowerPC Prozessor. In diesen Dateien wird konfiguriert, welche Versionen der Komponenten der zu erstellenden Toolchain verwendet und wohin die Ergebnisse installiert werden sollen.

Die Konfiguration erfolgt mittels folgender Umgebungsvariablen, die in dem Skript definiert werden:

TARBALLS_DIR:	Verzeichnis, in das crosstool den Quellcode abspeichert
RESULT_TOP:	Das Verzeichnis, in das die erstellte Toolchain installiert wird
GCC_LANGUAGES:	Programmiersprachen, die von der Toolchain unterstützt werden sollen

Welche Kombination von GCC und glibc verwendet werden, wird über die Zeilen festgelegt, die mit `eval` beginnen, z. B.:

```
eval `cat powerpc-405.dat gcc-3.4.1-glibc-2.3.3.dat` \
 sh all.sh --notest
```

Zeilen, die mit einem # beginnen sind auskommentiert und werden ignoriert. Das obige Beispielt verwendet GCC in der Version 3.4.1 und glibc 2.3.3 für eine PowerPC Plattform. Auf der Webseite von crosstool befindet sich auch eine Matrix, die Kombinationen für jede Plattform auflistet, von denen bekannt ist, dass sie zusammen eine funktionierende Toolchain ergeben.

Vorausgesetzt, der Benutzer, mit dem crosstool ausgeführt wird, hat Schreib- und Ausführrechte in den beiden konfigurierten Verzeichnissen, muss crosstool nicht als root ausgeführt werden, z. B.:

```
./demo-ppc405.sh
```

Die Ergebnisse dieses Erstellungsvorganges sind nach erfolgreichem Abschluss im Verzeichnis

```
/opt/crosstool/gcc-3.4.1-glibc-2.3.3/powerpc-405-linux-gnu
```

zu finden.

5.2.12 Scratchbox

Das Ziel von Scratchbox (http://www.scratchbox.org) ist es, Probleme, die insbesondere beim Cross-Compiling auftreten, zu minimieren. Für die Installation wird ein Linux-Rechner der x86 Plattform benötigt, auf dem idealerweise Debian als Distribution eingesetzt wird. Als Zielumgebung wird hauptsächlich die ARM-Plattform unterstützt.

5.2.13 Scratchbox 2

Eine distributionsunabhängige Weiterentwicklung von *Scratchbox* findet sich unter der Web-Adresse http://www.freedesktop.org/wiki/Software/sbox2. Der Hauptvorteil der Version 2 ist, dass für das Entwickeln mit *Scratchbox 2* keine root-Rechte benötigt werden.

5.2.14 OpenEmbedded

Das Projekt *OpenEmbedded* (http://www.openembedded.org) unterscheidet sich von den anderen beschriebenen Entwicklungsumgebungen insofern, als es den Schwerpunkt nicht auf die Benutzerschnittstelle legt, sondern auf automatisierte Erstellung einzelner Pakete oder ganzer Linuxsysteme für unterschiedliche Plattformen. OpenEmbedded ist aus dem Projekt *OpenZaurus* hervorgegangen, dessen Inhalt ein offenes Linuxsystem für den Sharp Zaurus PDA war. Inzwischen versteht sich OpenEmbedded als Metadatenspeicher und Entwicklungsumgebung mit dem beliebige Pervasive Linuxsysteme und einzelne Anwendungen definiert, konfiguriert und automatisiert generiert werden können.

OpenEmbedded besteht aus zwei zentralen Komponenten, dem bereits erwähnten Metadatenspeicher und dem Erstellungswerkzeug *BitBake* (Abschn. 5.2.10), wobei BitBake nicht als Teil von OpenEmbedded entwickelt wird. BitBake wertet die Metadaten aus und führt sämtliche Schritte durch, die zur Generierung der gewünschten Software notwendig sind. Dazu gehört das Herunterladen des benötigten Quellcode aus dem Internet, Einspielen möglicher Patches, Übersetzen mit einem Crosscompiler und Paketierung als installierbare Einheiten. Abbildung 5.2 veranschaulicht die Zusammenhänge. Für die Durchführung der Erstellungsschritte greift OpenEmbedded auf eine Reihe von Linuxwerkzeugen wie Python, GNU make, GNU patch, perl, GNU sed sowie eine Toolchain für die Zielplattform zurück, die auf dem Entwicklungsrechner installiert werden müssen. Die vollständige Liste der benötigten Softwareversionen mit Installationsanleitungen für zahlreiche Linuxdistributionen steht auf der Webseite von OpenEmbedded bereit.

5.2.14.1 Installation

Die beiden Komponenten von OpenEmbedded können von einem Linuxanwender in ein beliebiges Verzeichnis installiert werden, es werden keine Root-Rechte

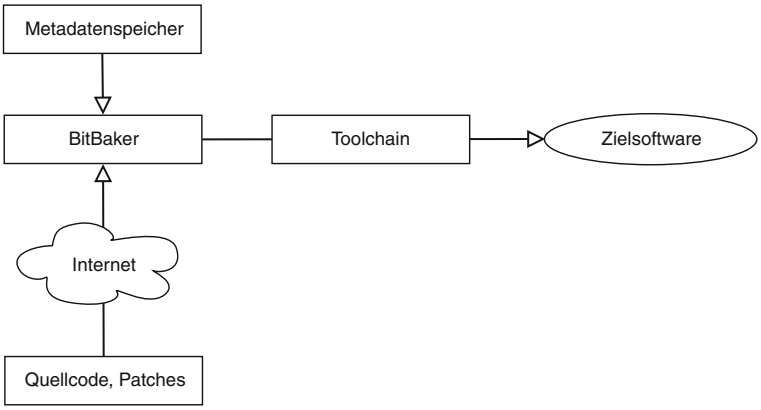

Abb. 5.2 Komponenten von OpenEmbedded

benötigt. Die einzige Einschränkung ist, dass der Pfad zu diesem Verzeichnis keine symbolischen Verweise enthalten darf. Für die folgenden Beispiele wird das Verzeichnis `/home/cc/oe-builds` verwendet. Als Shell wird *bash* eingesetzt.

Das OpenEmbedded Metadatenverzeichnis benutzt als Versionsverwaltungssystem die Software *Monotone*, welche auf der Webseite http://monotone.ca verfügbar ist. Zuerst muss eine Monotone-Datenbank in das Arbeitsverzeichnis heruntergeladen und entpackt werden. Anschliessend wird der Inhalt des Metadatenspeichers aus dem Netz aktualisiert und im nächsten Schritt aus der Datenbank in das Dateisystem extrahiert.

```
wget http://www.openembedded.org/snapshots/OE.mtn.bz2
bunzip2 -d OE.mtn.bz2
mtn --db=/home/cc/oe-builds/OE.mtn pull \
 monotone.openembedded.org org.openembedded.dev
mtn --db=/home/cc/oe-builds/OE.mtn checkout \
 --branch=org.openembedded.dev
```

5.2.14.2 Konfiguration

Für die Konfiguration steht die Datei `local.conf.sample` mit Beispieleinträgen zur Verfügung, die als Vorlage für die konkrete Umgebung gedacht sind, allerdings nicht als Kopie sondern eher als Dokumentation.

Zuerst wird ein Unterverzeichnis für die lokale Konfiguration angelegt und der Pfad für BitBake auf dieses Verzeichnis und die lokale Kopie von OpenEmbedded gesetzt. Die zentrale Konfigurationsdatei für BitBake, wie sie von OpenEmbedded benötigt wird, befindet sich im Verzeichnis `org.openembedded.dev/conf`.

```
cd /home/cc/oe-builds/
mkdir -p build/conf
export BBPATH=/home/cc/oe-builds/build:/home/cc/oe-builds/org.ope
nembedded.dev
```

Im Verzeichnis `/home/cc/oe-builds/build/conf` kann nun in der Datei `local.conf` über eine Reihe Schlüsselwort-Wert-Paare die lokale Konfiguration vorgenommen werden. Wichtig dabei ist, absolute Pfadangaben zu verwenden. Shell-Variablen wie $HOME werden von BitBake nicht expandiert, für Pfadangaben relativ zum Heimatverzeichnis des Benutzers steht die variable Abkürzung $HOME zur Verfügung. Zeilen dürfen nicht mit einem Leerzeichen beginnen und Zeilen, die mit einem „#“-Zeichen beginnen sind Kommentarzeilen.

`DL_DIR`	In dieses Verzeichnis werden die Dateien mit dem Quellcode aus dem Netz heruntergeladen.
`BBFILES`	Dieses Schlüsselwort spezifiziert, welche BitBake Dateien für den Erstellungsprozess verwendet werden sollen. BitBake Dateien haben die Dateiendung `.bb` und wurden während der Installation aus der Monotone Datenbank extrahiert.
`BBMASK`	Hier kann ein regulärer Ausdruck angegeben werden, um bestimmte BitBake Dateien explizit von der Verwendung im Erstellungsprozess auszunehmen.
`ASSUME_PROVIDED`	Falls eine bereits installierte Toolchain verwendet werden soll, kann das mit diesem Parameter konfiguriert werden, wobei zu beachten ist, dass die Werkzeuge der Toolchain über die Umgebungsvariable `$PATH` erreicht werden können.
`TMPDIR`	Das Standardverzeichnis für temporäre Dateien ist `/tmp` unterhalb des Installationsverzeichnisses von OpenEmbedded. Um ein anderes Verzeichnis anzugeben, wird das Kommentarzeichen am Anfang der Zeile entfernt und das gewünschte Verzeichnis eingetragen.
`MACHINE`	Hier wird das Gerät spezifiziert auf dem die erstellte Software eingesetzt werden soll, für einen HP iPaq 3900 z. B. `h3900`. Konfigurationsdateien für die von OpenEmbedded momentan unterstützen Geräte befinden sich in folgendem Unterverzeichnis unterhalb des OpenEmbedded Installationsverzeichnisses: `org.openembedded.dev/conf/machine` In diesen Dateien sind zahlreiche Konfigurationsparameter gesetzt, die während des Erstellungsprozesses ausgewertet werden.
`TARGET_ARCH`	Für den Fall, dass für das Zielgerät keine Konfigurationsdatei existiert und somit der Parameter `MACHINE` nicht spezifiziert werden kann, wird mit diesem Parameter die Prozessorarchitektur der Zielplattform festgelegt. Falls `MACHINE` sinnvoll belegt werden kann, wird dieser Wert nicht benötigt.

TARGET_OS	In den meisten Fällen wird hier `linux` angegeben, wenn das Zielsystem ein auf der Bibliothek *glibc* basierendes Linuxsystem ist, eventuell wird dieser Wert auch durch einen übergeordneten Parameter wie `DISTRO` oder `MACHINE` gesetzt. OpenEmbedded unterstützt aber auch Linuxsysteme, die die Bibliothek *uclibc* mit reduziertem Funktionsumfang und geringerem Speicherbedarf verwenden. In diesem Fall muss als Wert `linux-uclibc` eingetragen werden.
DISTRO	Wird auf dem Zielgerät eine spezielle Linuxdistribution wie z. B. Angstrom oder Openmoko eingesetzt, wird die entsprechende Distribution hier festgelegt. Konfigurationsdateien für Distributionen, die von OpenEmbedded unterstützt werden, befinden sich unterhalb des Installationsverzeichnisses in folgendem Verzeichnis: `org.openembedded.dev/conf/distro`.
KERNEL_VERSION	Für bestimmte Zielgeräte kann die zu verwendende Kernelversion konkretisiert werden. In den meisten Fällen wird dieser Parameter allerdings nicht verwendet und bleibt als Kommentarzeile oder wird gelöscht.
INHERIT	Wenn bei dem Parameter `DISTRO` eine Distribution angegeben wurde, werden die erstellten Komponenten in Installationspakete des von der Distribution verwendeten Formats gepackt. Andernfalls kann mit diesem Parameter konfiguriert werden, dass für die Ergebnisse des Erstellungsprozesses entsprechende Pakete erstellt werden. `package_ipk` und `package_tar` sind mögliche Werte für den Parameter.
IMAGE_FSTYPES	Wird ein komplettes Installationsabbild erstellt und nicht einzelne zu installierende Komponenten, müssen über diesen Wert mögliche zu verwendende Dateisystemtypen spezifiziert werden. Gültige Werte sind `jffs2`, `tar` und `cramfs`.
CACHE	Mit diesem Parameter wird der Cache-Speicher des Parsers deaktiviert. Üblicherweise wird der Parameter jedoch nicht genutzt und dementsprechend auch nicht gesetzt.
BBDEBUG	Wird dieser Parameter angegeben, produziert BitBake während des Erstellungsprozesses detaillierte Informationen über dessen Fortschritt. Das ist hilfreich, um Fehler im Erstellungsprozess zu finden.
DEBUG_BUILD	Mit diesem Wert wird BitBake angewiesen, beim Übersetzen so genannte Debuginformation in den Code einzubauen, die später verwendet werden kann, um Programmfehlern auf die Spur zu kommen.

`INHIBIT_PACKAGE_STRIP`	In Zusammenhang mit `DEBUG_BUILD` kann über diesen Parameter verhindert werden, dass Symbole aus erstellten Objektdateien entfernt werden.
`PROFILE_OPTIMIZATION`	Ebenso wie Debuginformation kann beim Übersetzen auch Profilinginformation in den erstellten Binärcode integriert werden. Diese Information kann zur Laufzeit von dem Profilingwerkzeug *gprof* ausgewertet werden, um das Laufzeitverhalten der Anwendung zu analysieren.
`SELECTED_OPTIMIZATION`	Über diesen Parameter werden die zu verwendenden Profiling-Einstellungen gesetzt.
`LDFLAGS`	Hiermit werden die Parameter zur Profiling-Konfiguration and die Liste der Linker-Parameter angehängt.
`PARALLEL_MAKE`	Falls der Erstellungsprozess auf einem Rechner mit mehreren Prozessoren bzw. Prozessorkernen durchgeführt wird, können Teile des Prozesses wie Übersetzen des Quellcodes auf mehrere Prozessoren verteilt und damit der Erstellungsprozess beschleunigt werden. Als Richtwerte gelten entweder die Anzahl der Prozessoren plus 1 oder die doppelte Prozessoranzahl.
`BBINCLUDELOGS`	BitBake protokolliert die einzelnen Schritte des Erstellungsprozesses in einer Logdatei mit. Für den Fehlerfall kann mit diesem Parameter bewirkt werden, dass das Logbuch angezeigt wird.
`CVS_TARBALL_STASH`	Hiermit wird eine Adresse angegeben, von der vorbereitete Archivdateien im TAR Format heruntergeladen werden können, was die für das Herunterladen des Quellcodes benötigte Zeit verkürzen kann. Wird der Parameter nicht angegeben, wird der Quellcode immer aus dem Versionsverwaltungssystem extrahiert.

Damit ist die Entwicklungsumgebung von OpenEmbedded vollständig konfiguriert und kann für die Erstellung von einzelnen Anwendungspaketen oder kompletter Installationsarchive eingesetzt werden. Beispielsweise erstellt der folgende Befehl ein Installationsarchiv für einen Taschenrechner in einer grafischen Opie-Umgebung (Abschn. 3.6.12):

```
bitbake opie-calculator
```

Um diesen Befehl auszuführen werden sämtliche Quellen und Übersetzungsinformationen aus dem Internet geladen und entsprechend ausgewertet. Das Ergebnis besteht nicht nur aus dem einsatzfähigen Installationsarchiv des

Programms, sondern auch aus sämtlichen Programmbibliotheken in der erforderlichen Version, von der die erstellte Software abhängig ist.

Die Installationsarchive befinden sich in dem Verzeichnis `deploy` unterhalb des Verzeichnisses, das in der Datei `local.conf` mit dem Wert `TMPDIR` konfiguriert wurde.

Für dieses Beispiel wurde die Datei

```
/tmp/oetmp/deploy/ipk/opie-calculator\_1.2.0+cvs-20050918-r1\_arm
.ipk
```

erstellt. Diese kann nur auf das Zielgerät übertragen und installiert werden.

5.2.15 Poky

Das Projekt *Poky*, zu finden unter http://pokylinux.org/, ermöglicht den Aufbau von auf bestimmte Geräte zugeschnittenen Linux-Systemen, es handelt sich also um eine Build-Umgebung, mit der eigene Linux-Distributionen bzw. Umgebungen erstellt werden können.

Die Poky-Build-Umgebung basiert auf OpenEmbedded (Abschn. 5.2.14) und wird wiederum vom Yocto Project (Abschn. 5.2.16) eingesetzt. Für die Erstellung von Linux- Abbildern für diverse mobile Geräte greift das Yocto Project teilweise auf existierende quelloffene Standards wie X11, Matchbox, GTK+, Pimlico, Clutter und GNOME Mobile zurück.

Die von Poky erstellten Abbilder können innerhalb von Qemu gestartet werden, was den Einsatz für das Pervasive Computing besonders interessant macht.

5.2.16 Yocto Project

Unter dem Dach der Linux Foundation wird das *Yocto Project* betrieben, zur Vereinheitlichung von Werkzeugen, Methoden und Vorlagen zur Entwicklung von Linux-basierten Systemen und Produkten. Die Homepage des Projektes ist http://www.yoctoproject.org/. Das Yocto Project beinhaltet u.a. die Poky Build-Umgebung (Abschn. 5.2.15), Emulatoren und die Integration in diverse grafische Entwicklungsumgebungen.

Bei der Build-Umgebung setzt das Yacto Project auf OpenEmbedded (Abschn. 5.2.14), beide Projekte arbeiten eng zusammen.

Für das erstellt Linux-System wird auch eine passende Toolchain erstellt.

Um ein neues System zu erstellen, werden zunächst die Projektdateien heruntergeladen und extrahiert. Anschliessend wird die Datei `poky-init-build-env` als Skript ausgeführt, um die Umgebung zu initialisieren. Der zweite Parameter ist der Name, der für das Build-Verzeichnis verwendet werden soll. In das Build-Verzeichnis werden sämliche Ergebnisse des Erstellungsprozesses abgelegt. Als Standardwert wird `build` angenommen.

```
wget http://www.yoctoproject.org/downloads/poky/poky-laverne-4.0.
tar.bz2
tar xjf poky-laverne-4.0.tar.bz2
source poky-laverne-4.0/poky-init-build-env poky-build
```

Im Build-Verzeichnis wird noch ein Unterverzeichnis `conf` angelegt, in dem sich die Konfigurationsdatei `local.conf` für OpenEmbedded befindet. In dieser Datei wird z. B. auch festgelegt, welche Hardware-Architektur zum Einsatz kommen soll.

Mit dem Befehl `bitbake` kann nun ein Linux-System erstellt werden, die benötigten `.bb`-Dateien befinden sich unterhalb des `meta`-Verzeichnisses.

```
bitbake poky-image-sato
```

Dieser Bitbake Befehl erzeugt das Referenzsystem mit dem Grafiksystem *Sato*. Der komplette Erstellvorgang ist ein sehr zeitaufwändiger Prozess.

5.2.17 Terminal-Emulation

Eine Terminal-Emulation ist in den allermeisten Fällen unverzichtbar, wenn es darum geht, sich über eine serielle Schnittstelle (Abschn. 4.2) auf dem Linux-System der Zielumgebung anzumelden und dort mit einer Kommandozeile zu arbeiten.

5.2.17.1 Minicom

Das Programm *Minicom* wird von den meisten Linux-Distributionen vorinstalliert. Mit ihm lassen sich Kommunikationsverbindungen über eine Modem, bzw. eine serielle Schnittstelle aufbauen. Falls an die zu verwendende Schnittstelle unterschiedliche Geräte angeschlossen werden, können die jeweiligen Verbindungsinformationen sehr einfach in einer Konfigurationsdatei abgelegt werden. Standardmässig, d.h. wenn keine Konfiguration beim Starten angegeben wird, verwendet Minicom die Datei `minirc.dfl`, meistens im Verzeichnis `/etc` oder `/etc/minicom`. Die Dateiendung entspricht dem Namen der Konfiguration, z. B. kann in einer Datei mit dem Namen `minirc.arm` die Verbindung zu einer ARM-Zielumgebung abgelegt und mit `minicom arm` verwendet werden. Die Projektseite im Internet ist http://alioth.debian.org/projects/minicom.

Zur Konfiguration einer Verbindung dient der Parameter `-s`, in diesem Fall springt Minicom direkt in das Konfigurationsmenü, ohne beim Start die Schnittstelle zu initialisieren.

Nach dem Start von Minicom können über die Tastenkombination `Strg-A` diverse Befehle an Minicom geschickt werden, über `Strg-A Z` wird eine Übersicht zentraler Kommandos angezeigt.

5.2.17.2 Kermit

Nach einer Figur aus der Muppet-Show benannt, ist *Kermit* eine Alternative zu minicom als Terminal-Emulation. Kermit (http://www.columbia.edu/kermit) wurde 1981 an der Columbia Universität in New York entwickelt und wird dort seit diesem Zeitpunkt gepflegt. Es besteht aus einem eigenen Dateiübertragungsprotokoll sowie einer Reihe Programme und wurde für zahlreiche Betriebssysteme portiert. In der Version C-Kermit steht Kermit für UNIX und damit auch für Linux zur Verfügung.

Im Unterschied zu Minicom bietet Kermit auch die Möglichkeit, eine Terminalsicht über eine Netzwerkverbindung aufzubauen und unterstützt Netzwerkübertragungsprotokolle wie *FTP* und *HTTP*, sowie die Anmeldung auf einem entfernten System über *Telnet* oder `Rlogin`.

Kermit wird mit dem Befehl `kermit` gestartet und befindet sich danach in einem interaktiven Modus. Mit folgender Befehlsfolge baut Kermit eine einfache Terminal-Verbindung über die serielle Schnittstelle `/dev/ttyS0` auf:

```
set line /dev/ttyS0
set speed 115200
set parity even
connect
```

Dabei wird zunächst die zu verwendende Schnittstelle gesetzt und anschliessend die Übertragungsgeschwindigkeit. Das Paritätsbit wird so eingestellt, dass die Anzahl an Bits, die mit jedem Zeichen übertragen werden, immer eine gerade Zahl ergibt (Abschn. 4.2). Mit dem Befehl `connect` baut Kermit die Verbindung zur Zielumgebung auf.

Sämtliche im interaktiven Modus verfügbaren Befehle können auch in eine Datei geschrieben und von Kermit als Skript ausgeführt werden.

5.2.18 Xnest

Xnest ist ein X-Window-Server innerhalb eines X-Fensters. Damit kann die Oberfläche eines Programms, das auf einem anderen System ausgeführt wird, in einem Fenster des Entwicklungssystem angezeigt werden. Ein Teil des Framebuffers (Abschn. 3.2.2) des Systems, auf dem Xnest läuft, wird in ein X-Fenster für die Darstellung abgebildet. Xnest bietet somit einen virtuellen Framebuffer durch die Nutzung des Framebuffers des Host-Systems an und nutzt für die Darstellung das X-System der Host-Umgebung. Das ist hilfreich für die Entwicklung von Programmen für das X-Window System, da man damit die grafische Oberfläche des Zielsystems auf dem Entwicklungsrechner darstellen kann, ohne auf die Ausgabe auf dem Zielsystem angewiesen zu sein. Ausserdem lassen sich auf diese Weise unterschiedliche Auflösungen der Anzeigergeräte von Zielsystemen testen.

Üblicherweise verwendet der X-Server des Host-Systems für die Ausgabe die Anzeige mit der Nummer `0`. In diesem Fall kann normalerweise mit der Nummer `1` für die Vergabe einer virtuellen Anzeige begonnen werden.

```
Xnest :1 -geometry 800x600
```

Der Aufruf erzeugt ein neues X-Fenster, in dem ein X-Server mit der Nummer `1` und einer Auflösung von 800 auf 600 Punkten gestartet wird. Nun kann die Anzeige des Zielsystems auf diesen X-Server umgeleitet werden.

5.2.19 Xephyr

Xephyr ist ein X-Window-Server, der wie Xnest ein Fenster eines anderen Systems als Framebuffer verwendet. Entwickelt wird Xephyr im Rahmen des Freedesktop-Projektes, auf http://freedesktop.org/wiki/Software/Xephyr findet sich die Projektseite. Xephyr stellt eine eigenständige Entwicklung eines X-Servers dar und ist nicht auf die Implementierung des Host-Systems angewiesen. Daher kann Xephyr Erweiterungen des X-Window Systems anbieten, die das Host-System selbst möglicherweise nicht unterstützt.

```
Xephyr :1 -screen 800x600
```

Das Ergebnis dieses Aufrufs ist ein neues X-Fenster, in dem Xephyr die Anzeige mit der Nummer `1` mit einer Auflösung von 800 auf 600 Punkten verwaltet.

5.3 Virtualisierung

Unter *Virtualisierung* versteht man im hier relevanten Zusammenhang das Loslösen des Betriebssystems von einer konkreten Ziel-Hardware unter Bereitstellung eines künstlichen Umfelds, in dem das Betriebssystem möglichst unverändert betrieben werden kann. Konkret bedeutet Virtualisierung hier ein Stück Software, das die Zielumgebung, also ein Stück Hardware nachbildet und emuliert. Hardware-Emulatoren sind Programme, die eine bestimmte Hardwareumgebung in Form von Software nachbilden. Damit kann die gewünschte Zielumgebung verwendet werden, ohne dass tatsächliche Hardware verfügbar sein muss. Das hat zum einen den Vorteil, dass sowohl die Entwicklungsumgebung als auch die emulierte Zielumgebung auf dem selben Rechner benutzt werden können, was die Kommunikation mit der Zielumgebung erleichtern kann. Zum anderen kann die Hardware mit unterschiedlichen emulierten Bestandteilen erstellt werden, z. B. kann der Umgebung für Entwicklungszwecke ein grösserer Arbeitsspeicher zugewiesen werden, als auf der tatsächlichen Umgebung vorhanden sein wird. Auf diese Weise können unterschiedliche Hardwarekonfigurationen auf einfache Weise getestet werden.

Allerdings kann im Entwicklungsprozess auf physikalische Hardware nicht ganz verzichtet werden, da sich emulierte und tatsächliche Hardware nie vollständig identisch verhalten, auch wenn die Unterschiede nicht gross ein müssen.

Das Fehlen echter Hardware hat z. B. auch zur Folge, dass Eingaben durch den Benutzer nicht über eine reale Tastatur oder einen berührungsempfindlichen

Bildschirm, sondern mit Hilfe alternativer Geräte wie einer Maus auf dem emulierten Gerät getätigt werden. Mit einem pixelgenauen Zeiger einer Maus, lassen sich die entsprechenden Schaltflächen auf der Benutzeroberfläche deutlich präziser ansteuern, als beispielsweise mit Fingern auf einem berührungsempfindlichen Bildschirm. Derartiger Unterschiede zwischen dem tatsächlichen Gerät und der Emulationssoftware muss man sich bei der Entwicklung bewusst sein.

5.3.1 QEMU

QEMU ist ein quelloffener Hardware-Emulator, der zahlreiche Prozessoren unterstützt und zwar sowohl als Host- als auch als Gastsystem. Im Internet ist QEMU unter http://wiki.qemu.org zu finden. Für die Emulation der Zielumgebung wird eine so genannte *Virtuelle Maschine* (*VM*) erstellt. Falls nicht bereits fertige Abbilder für die gewünschte Zielplattform existieren, kann QEMU auch entsprechend konfiguriert und übersetzt werden.

Der innhalb QEMU ausgeführten Umgebung kann Zugriff auf Ressourcen des Entwicklungssystems gegeben werden. Beispielsweise kann ein von QEMU ausgeführter Prozess Zugriff auf USB-Geräte oder Speichermedien erhalten.

QEMU kann auf zweierlei Weise verwendet werden. Bei der Emulation eines vollständigen Systems (engl.: *Full System Emulation*) wird der Prozessor der Zielumgebung mitsamt mehr oder weniger umfangreichen Zusatzgeräten, wie Netzwerkkarte, Soundkarte, Grafikkarte etc. abgebildet, um ein System zu erhalten, das möglichst genau der Hardware-Realität entspricht. Bei *User Mode Emulation* dagegen wird ein Programm, das für eine andere Hardware-Plattform übersetzt wurde, in einem Prozess der Entwicklungsumgebung ausgeführt.

5.3.1.1 Full System Emulation

Für die Emulation einer Systemkonfiguration stehen pro Prozessorarchitektur entsprechend angepasste QEMU-Implementierungen bereit, die folgender Aufrufkonvention folgen: `qemu-system-<architektur>`. Beispielsweise startet der Befehl `qemu-system-ppc` die QEMU-Version für Umgebungen, die auf der PowerPC-Prozessorarchitektur basieren.

Für den Start des Zielsystems benötigt QEMU abhängig vom Zielsystemtyp ein Festplattenabbild (engl.: Disk Image) des zu startenden Systems oder den zu startenden Kernel und initiale RAM-Disk. Auf der Webseite von QEMU stehen Beispiele für alle unterstützten Hardware-Architekturen zum Herunterladen bereit. Das Projekt *FreeOsZoo* hat es sich zur Aufgabe gemacht, auf der Webseite http://www.oszoo.org QEMU Abbilder möglichst vieler frei verfügbarer Betriebssysteme zur Verfügung zu stellen.

Nach der Installation kann die Zielumgebung bspw. für eine ARM-Plattform gestartet werden.

```
qemu-system-arm -kernel ~/arm-test/zImage.integrator \
 -initrd ~/arm-test/arm_root.img
```

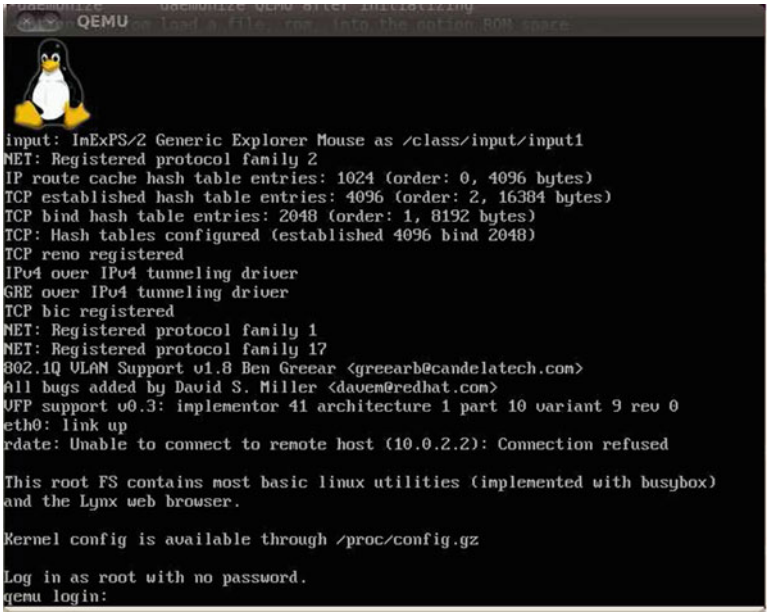

Abb. 5.3 Der QEMU Emulator

Der nächste Schritt ist nun, Daten mit der Zielumgebung auszutauschen. Dazu gibt es verschiedene Möglichkeiten.

QEMU bietet diverse Alternativen, mit der Zielumgebung über ein virtuelles Netzwerk zu kommunizieren, bzw. sie in das Netzwerk der Hostumgebung zu integrieren.

5.3.1.2 User Mode Emulation

Die Aufrufkonvention von QEMU für das Ausführen von Programmen mit dem User Space Emulator ist wie folgt: `qemu-<architektur>`. Zusätzlich benötigt QEMU noch das auszuführende Programm und die verwendeten dynamischen Bibliotheken inkl. `glibc`.

5.4 Integrierte Entwicklungsumgebungen

Integrierte Entwicklungsumgebungen (engl.: Integrated Development Environment – IDE) haben den Anspruch, Software-Entwicklung zu vereinfachen und zu beschleunigen, indem alle benötigten Werkzeuge wie Editor, Übersetzer, Linker oder Debugger über eine einheitliche, zentrale Oberfläche bedient werden.

Es gibt eine grosse Anzahl an Integrierten Entwicklungsumgebungen für Linux und andere Betriebssysteme, frei verfügbare und kommerzielle, die diverse Programmiersprachen und Zielplattformen unterstützen. In diesem Kapitel wird

beispielhaft die populäre Open Source Entwicklungsumgebung *Eclipse* vorgestellt, die auch für Linux verfügbar ist. Alternativen sind z. B. *KDevelop* (http://www.kdevelop.org/) für auf KDE basierende Oberflächen und die in die GNOME-Oberfläche integrierte Entwicklungsumgebung *Anjuta DevStudio* (http://projects.gnome.org/anjuta).

5.4.1 Eclipse

Eclipse (http://www.eclipse.org) ist an sich zunächst keine Entwicklungsumgebung sondern eine Anwendungsplattform. Der erste Anwendungsbereich, für den die Eclipse Plattform genutzt wurde und für den sie auch ursprünglich entwickelt wurde, war aber eine integrierte Entwicklungsumgebung für Java Anwendungen.

Das Eclipse Projekt geht auf eine Initiative der Firma IBM zurück, in deren Rahmen die erste Version von Eclipse als Open Source veröffentlicht wurde. Um die Weiterentwicklung unabhängig von einem bestimmten Unternehmen zu ermöglichen und zu fördern, wurde das Eclipse Konsortium gegründet, dem Unternehmen beitreten können, die die Eclipse Plattform für kommerzielle Softwareprodukte nutzen möchten. Inzwischen wurde das Eclipse Konsortium in eine gemeinnützige Organisation, die *Eclipse Foundation* umgewandelt.

Seit der Veröffentlichung des Quellcode im Jahr 2001 hat das Eclipse Projekt einen wahren Siegeszug angetreten, und immer mehr Unternehmen beteiligen sich an der Weiterentwicklung. Die Attraktivität von Eclipse kann neben der Verfügbarkeit als Open Source mit der grossen Flexibilität der Architektur begründet werden. Eclipse ist fast vollständig in der Programmiersprache Java geschrieben und daher sehr leicht auf ein bestimmtes Betriebssystem zu portieren. Die eigentliche Anwendungsfunktionalität wird über das so genannte Plug-In-Konzept realisiert. Plug-Ins bedienen eine von Eclipse definierte Schnittstelle und erweitern damit die über die Plattform verfügbare Funktionalität. Der Zugriff auf ein Plug-In erfolgt für den Benutzer für alle Plug-Ins auf die gleiche Weise. Die Anzahl frei verfügbarer oder kommerzieller Plug-Ins wächst ständig. Auch viele Hersteller von Werkzeugen für die Entwicklung von Software für das Pervasive Computing stellen ihre Produkte inzwischen als Plug-Ins für Eclipse bereit.

Das Anwendungskonzept von Eclipse basiert auf so genannten *Perspektiven* (engl.: Perspectives) sowie *Sichten* (engl.: Views) und *Editoren*. Für einen bestimmten Aufgabenbereich, wie z. B. die Entwicklung von Java oder C++ Anwendungen gibt es jeweils eine eigene Perspektive, die sich wiederum in diverse Sichten und Editoren gliedert. Eine Sicht entspricht einem Fenster innerhalb der Eclipse Arbeitsoberfläche, in dem ein bestimmter Teil der gerade genutzten Perspektive dargestellt wird. Mit einem Editor können für den Themenbereich der gerade genutzen Perspektive relevante Inhalte bearbeitet werden.

Abbildung 5.4 zeigt beispielhaft den Aufbau der Eclipse Oberfläche in der C/C++ Perspektive. Selbstverständlich kann die Oberfläche, also die Grösse, Anordnung und Position von Sichten und Editoren durch den Benutzer individuell angepasst werden.

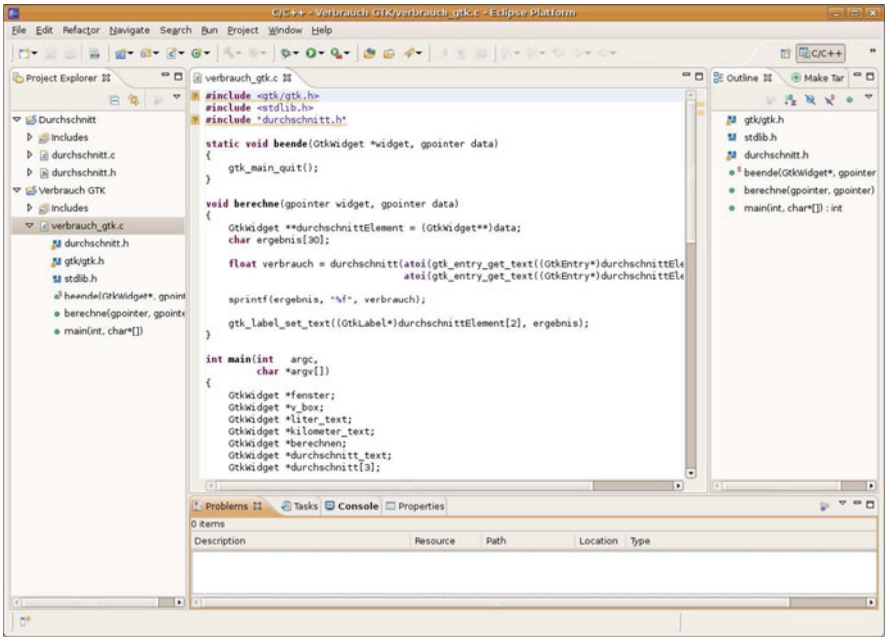

Abb. 5.4 Die Eclipse Oberfläche

5.4.1.1 Das Plug-In Konzept

Für die Erweiterung der Funktionalität und die Entwicklung von Plug-Ins spezifiziert Eclipse so genannte *Erweiterungspunkte* (*engl.: Extension Points*). Plug-Ins müssen die Vorgaben dieser Erweiterungspunkte erfüllen, d.h. benötigte Informationen bereitstellen, und sie können auch selbst solche Erweiterungspunkte definieren, die dann wiederum von anderen Plug-Ins erweitert werden können. Die Spezifikation sowohl von Erweiterungspunkten als auch Erweiterungen erfolgt im XML-Format (Abschn. 3.5.2.1).

Plug-Ins werden von der Eclipse Plattform dynamisch je nach Bedarf geladen. Die Deklaration eines Plug-Ins erfolgt über eine so genannten Manifest-Datei mit dem Dateinamen `MANIFEST.MF`. Ein Plug-In deklariert in einer Plug-In Konfigurationsdatei, welche Erweiterungspunkte es erweitert, bzw. welche es definiert. Dies ist eine Datei im XML Format mit dem Dateinamen `plugin.xml`.

Wie die Eclipse Plattform selbst werden Plug-Ins in der Java Programmiersprache entwickelt. Die Klasse, über die ein Plug-In von der Plattform in den Arbeitsspeicher geladen und angesprochen wird, der so genannte Aktivator, wird in der Manifest-Datei des Plug-Ins deklariert. Manche Plug-Ins kommen auch ohne eigenen Java-Code aus. In solchen Fällen werden sämtliche für das Plug-In benötigte Informationen in entsprechenden Erweiterungen in der Manifest- und der Konfigurationsdatei spezifiziert.

Für die Entwicklung von Plug-Ins stellt Eclipse eine eigene Perspektive bereit, die u.a. einen Editor zur Bearbeitung der beiden Dateien `MANIFEST.MF` und `plugin.xml` beinhaltet. Die für die Entwicklung von Eclipse Plug-Ins benötigten Plug-Ins, zusammengefasst in der Plug-In Entwicklungsumgebung (engl.: Plug-In Development Environment – PDE), sind in der Softwareentwicklungsversion (engl.: Software Development Kit – SDK) von Eclipse enthalten. Um ein selbst entwickeltes Plug-In zu testen, wird aus der Eclipse Entwicklungsumgebung heraus, in der das Plug-In entwickelt wurde, eine zweite Entwicklungsumgebung gestartet, in der das neue Plug-In installiert ist. Dieses Vorgehen macht die Entwicklung von Plug-Ins sehr effizient.

5.4.1.2 Aktualisierung der Umgebung

Für die Verteilung und Installation von zusätzlicher Funktionalität und die Aktualisierung bereits installierter Plug-Ins hat Eclipse das Konzept so genannter Aktualisierungsorte (engl.: Update Site) übernommen. Dazu werden im Internet an definierten Adressen Plug-Ins mit einem definierten Versionsstand abgelegt. Die Adressen dieser Update-Sites werden in einer Eclipse-Installation unter dem Menüpunkt *Help* → *Software Updates...* verwaltet. Über denselben Dialog wird dann auch die Installation bzw. Aktualisierung von Plug-Ins durchgeführt.

5.4.1.3 C/C++ Development Tools (CDT)

Ein insbesondere für die Entwicklung von Software für das Pervasive Computing interessantes Eclipse Projekt sind die *CDT* mit dem Ziel, eine vollständige Entwicklungsumgebung für die Programmiersprachen C und C++ bereitzustellen. Die CDT können von der Seite http://www.eclipse.org/cdt/ heruntergeladen werden.

Die CDT beinhalten u.a. Parser für die genannten Sprachen sowie komfortable Editoren und weitere Werkzeuge für die Entwicklung mit C und C++. Die CDT bieten die Möglichkeit, über Erweiterungspunkte beliebige C und C++ Werkzeuge wie Übersetzer, Linker etc. einzubinden.

Wie für die Eclipse Plattform selbst, gibt es von den CDT zwei Versionen, eine für die Entwicklung von Programmen für die Plattformen, für die bereits fertige Plug-Ins vorliegen, die die Erweiterungspunkte der CDT entsprechend erweitern, und eine SDK-Version, die die Entwicklung eben dieser Plug-Ins für beliebige C und C++ Werkzeuge und Plattformen unterstützt.

Um eine zusätzliche Toolchain in die CDT einzubinden, muss für diese ein neues Plug-In installiert werden, das die Aufrufregeln und Konfigurationsparameter der Toolchain enthält. Um dieses Plug-In zu entwickeln, wird die SDK-Version der CDT benötigt, da diese die Definition des erforderlichen Erweiterungspunkts bereitstellt:

```
org.eclipse.cdt.managedbuilder.core.buildDefinitions
```

Im Folgenden werden die Prinzipien der Definition eines CDT Plug-Ins für eine zusätzliche Toolchain für ARM-Plattformen erläutert. Anschliessend kann Eclipse

als Entwicklungsumgebung für C und C++ Programme dieser Plattform eingesetzt werden.

Für dieses Beispiel wird die in Abschn. 5.1.2 erstellte Toolchain verwendet, die auf dem Entwicklungsrechner installiert ist. Die eingesetzte Version von Eclipse ist 3.4.0 mit den CDT in der Version 5.0.0. Abhängig von der vorhandenen Eclipse-Installation muss noch die Update-Site des CDT-Projektes konfiguriert werden, um die benötigten Plug-Ins zu installieren. Die grundlegenden Konzepte haben sich auch in der aktuellen Version 3.7 nicht geändert. Insgesamt achtet die Eclipse-Entwicklergemeinde stark auf Kompatibilität von Schnittstellen über Versionen hinweg.

Um nun die zusätzliche Toolchain zu integrieren, muss ein neues Plug-In-Projekt angelegt werden, in dem die Bestandteile der Toolchain konfiguriert werden. Im folgenden Assistentenfenster wird in der Kategorie *Plug-in Development* der Projekttyp *Plug-in Project* ausgewählt (Abb. 5.6). Auf der nächsten Seite des Dialogs wird der Projektname angegeben und die Schaltfläche *Create a Java*

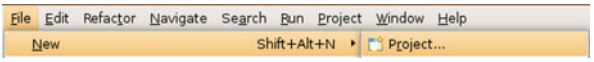

Abb. 5.5 Menüzeile zur Projekterstellung

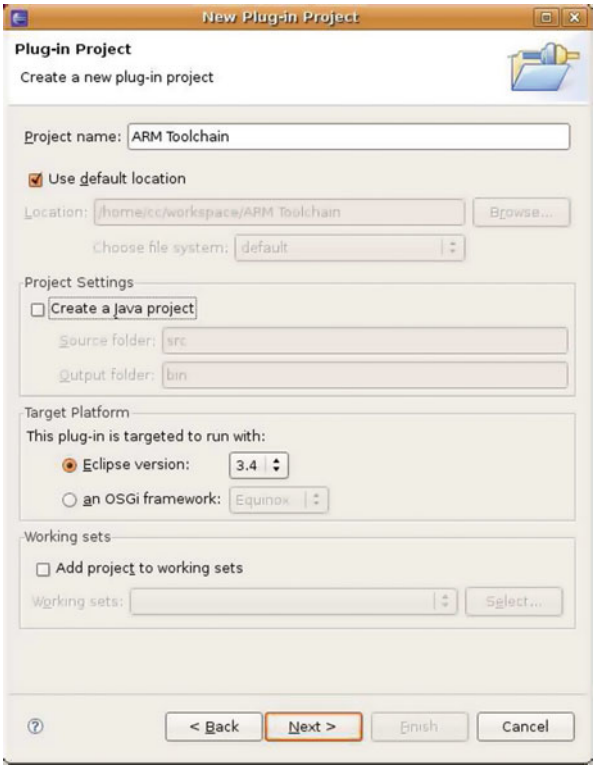

Abb. 5.6 Dialog zur Erstellung eines neuen Plug-In Projektes

project deaktiviert, da für die Konfiguration kein Java Code entwickelt werden muss, die übrigen Einstellungen auf dieser Seite können unverändert übernommen werden. Ausserdem sollte auf der nächsten Seite die voreingestellte Plug-In ID geändert werden. Allen Elementen innerhalb einer Plug-In Definition werden eindeutige Identifikationsnummern (ID) zugewiesen, so dass sie innerhalb der Konfiguration referenziert werden können. Die Konvention für die Vergabe von IDs orientiert sich an dem aus den Java-Umfeld bekannten Paketnamen: Die Bestandteile des Internet-Domänennamens der verantwortlichen Organisation, z. B. `springer.com`, werden in umgekehrter Reihenfolge an den Anfang gestellt. Anschliessend können beliebige weitere Elemente zur genaueren Klassifizierung, wie z. B. Abteilungs- oder Projektname, jeweils durch Punkt getrennt angehängt werden. Abschliessend folgt der das jeweilige Element bezeichnende eindeutige Bestandteil. Für dieses Beispiel wird die ID `com.springer.arm.toolchain` verwendet. Damit sind ausreichend Informationen für die Projekterstellung vorhanden und das Projekt kann über den Auswahlknopf *Finish* generiert werden. Für das Plug-In wurde u.a. die benötigte Manifest-Datei `MANIFEST.MF` erstellt, in der die Informationen abgelegt sind, die von Eclipse benötigt werden, um das Plug-In in die Eclipse-Plattform zu integrieren. Nach dem Erstellen des Plug-Ins öffnet sich der Manifest-Editor in der Perspektive für Plug-In Entwicklung, mit dem die Datei `MANIFEST.MF` bearbeitet werden kann.

Das neue Plug-In benötigt die Basisfunktionalität der CDT, u.a. um den genannten Erweiterungspunkt referenzieren zu können. Diese Abhängigkeit muss im Manifest-Editor auf der Seite *Dependencies* explizit gemacht werden, indem das Plug-In `org.eclipse.cdt.managedbuilder.core` über die Schaltfläche *Add...* hinzugefügt wird. Nachdem nun die Inhalte dieses Plug-Ins zugreifbar gemacht wurden, kann auf der Seite *Extensions* eine Implementierung des Erweiterungspunkts `buildDefinitions` deklariert werden, sowie eine Identifikationsnummer, z. B. `com.springer.arm.buildDefinition`, und ein verständlicher Name für die Erweiterung angegeben werden. Mit *verständlichen Namen* sind Bezeichnungen der beschriebenen Konfigurationselemente gemeint, die einem Benutzer des Systems an der Benutzeroberfläche angezeigt werden. Durch diese Angaben wird Eclipse bekannt gemacht, dass die CDT um zusätzliche Funktionalität erweitert werden soll.

Um Projekte und deren Bestandteile mit der neuen Toolchain erstellen zu können, benötigen die CDT Informationen über die zu verwendenden Werkzeuge sowie deren Ein- und Ausgabeformate und Abhängigkeiten, z. B. die Entstehungsreihenfolge der Erstellungsergebnisse. Der Erweiterungspunkt `buildDefinitions` definiert dafür eine Hierarchie von Elementen, die diese Informationen aufnehmen und miteinander kombinieren. Die wichtigsten sind:

tool: Werkzeuge (engl.: Tool) im Sinne der CDT sind die Bestandteile der Toolchain, die Quellcode und Zwischenergebnisse von einem oder mehreren Eingabetypen (engl.: Input Type) in einen Ausgabetyp (engl.: Output Type) überführen, wie z. B. Übersetzer

	oder Linker. Werkzeuge werden über Optionen (engl.: Option) parametrisiert. Optionen wiederum können in Kategorien (engl.: Category) logisch zusammengefasst werden. Optionen, Optionskategorien, Eingabe- und Ausgabetypen werden als Bestandteilelemente von Werkzeugelementen definiert.
projectType:	Projekttypen (engl.: Project Type) beinhalten Konfigurationselemente, um ein bestimmtes Entwicklungsergebnis zu generieren, z. B. ausführbare Dateien oder Programmbibliotheken. Für einen Projekttyp muss mindestens eine Konfiguration definiert werden.
configuration:	Eine Konfiguration (engl: Configuration) beinhaltet Informationen, um eine bestimmte Ausprägung des Projekttyps zu erstellen, beispielsweise könnte jeweils eine Konfiguration für Debug-Version und eine Produktionsversion existieren.
toolChain:	Das Toolchain-Element als Teil einer Konfiguration gruppiert die eingesetzten Werkzeuge und deren Optionen. Mit Hilfe dieses Elements können unterschiedliche Toolchains zur Erstellung des gleichen Projektergebnisses eingesetzt werden.

Zunächst werden zwei der zu integrierenden Werkzeuge spezifiziert, der Übersetzer und der Linker. Die Definition eines Werkzeugs wird angelegt, indem auf der Seite *Extensions* des Manifest-Editors mit der rechten Maustaste auf die bereits bestehende Erweiterung geklickt wird, um das Kontextmenü zu öffnen. Über den Menüpunkt *New* → *tool* wird der Eintrag für das neue Werkzeug angelegt und mit Identifikationsnummer und Namen versehen.

Zusätzlich zu Identifikationsnummer und Name muss hier über `natureFilter` angegeben werden, ob für C und C++ Programme unterschiedliche Übersetzer verwendet werden. In dem hier beschriebenen Beispiel ist der GCC Übersetzer für C und C++ Programme gültig, deshalb wird `both` aus der Liste ausgewählt. Wie bei der GCC üblich wird die Ausgabedatei mit dem Parameter `-o` angegeben, dieser Wert wird dementsprechend im Feld `outputFlag` eingetragen. Der Übersetzer selbst wird in diesem Beispiel mit dem Befehl `arm-linux-gcc` aufgerufen. Dieser Befehl wird im Feld `command` angegeben.

Als Eingabetypen sollen sowohl Quelldateien der Sprachen C und C++ als auch die dazugehörenden Header-Dateien dienen. Zunächst wird über das Kontextmenü der `tool`-Erweiterung (*New* → *inputType*) ein Eingabetyp für C-Header-Dateien definiert. Neben den obligatorischen ID- und Namen-Feldern werden für dieses Element im Feld für die Quelldateien `sources` die Dateiendungen `h,H` spezifiziert. Dass mehr als eine Datei als Eingabe für den Übersetzer dient, wird durch Auswahl von `true` im Feld `multipleOfType` klargestellt.

Wie für die Header-Dateien wird innerhalb des `tool`-Elements noch ein Element für die Quelldateien in der Sprache C benötigt. Die Definition des Elements erfolgt analog zu der für die Header-Dateien mit dem Unterschied, dass diesmal die Dateiendungen `c,C` im Feld `sources` spezifiziert werden. Zusätzlich wird noch die Abhängigkeit zu den Header-Dateien im Feld `dependencyContentType` angegeben, da C Quelldateien von Header Dateien abhängen können, sowie mit

dem Parameter `primaryInput` festgelegt, dass es sich hier um die Haupt-Eingabedateien für den Übersetzer handelt.

Ebenso wie die Definitionen für Quell- und Header-Dateien von C-Programmen, sollten entsprechende Pendants für die Sprache C++ erstellt werden.

Das Ausgabeformat für den Übersetzer ist eine Objektdatei, für die ein entsprechendes Ausgabetyp-Element definiert werden muss. Hier ist die Dateiendung `o` im Feld `outputs` zu spezifizieren, sowie beispielsweise `OBJS` als Variable für Ergebnisse im Build-Prozess anhand des Parameters `buildVariable`. Über diese Build-Variable werden die erstellten Ausgabedateien später in der Build Datei `Makefile` referenziert und können so als Eingabedaten für darauffolgende Erstellungsschritte dienen.

Um das Werkzeug, in diesem Beispiel der Übersetzer `arm-linux-gcc`, dazu zu bringen, nur zu übersetzen und nicht den Linkvorgang durchzuführen, wird eine entsprechende Option für den Aufruf des Werkzeugs über ein `option` Element innerhalb des `tools` Elements definiert. Der `valueType` für dieses Beispiel ist `string` und die Option `-c`, anzugeben im Feld `defaultValue`.

Das Linker-Werkzeug für dieses Beispiel ist ebenfalls `arm-linux-gcc`, daher erfolgt die Definition des `tool`-Elements analog zu der des Übersetzers bis auf die Werte für ID und Name.

Als Eingabetyp dient der für den Übersetzer definierte Ausgabetyp, ebenfalls mit dem Wert `OBJS` für den Parameter `buildVariable`. Als Quelldateityp wird `o` im Feld `sources` angegeben. Sowohl `primaryInput` als auch `multipleOfType` werden auf `true` gesetzt, da die vom Übersetzer produzierten Objektdateien die Haupteingabedaten für den Linker darstellen und mehrere Objektdateien zusammen in einem Schritt verarbeitet werden können. Für den Ausgabetyp muss nur das entsprechende Element angelegt werden, da ausführbare Dateien unter Linux keine Dateiendung besitzen, benötigt dieses Element im ersten Schritt keine zusätzlichen Konfigurationsinformationen.

Als nächstes werden die *Projekttypen* definiert, die mit dem neuen Plug-In erzeugt werden sollen. Ein Projekttyp beschreibt, welche Arbeitsergebnisse mit diesem Projekt erstellt werden, wie z. B. ausführbare Dateien oder Programmbibliotheken, und stellt die Toolchain zusammen, die für den Entwicklungsprozess benötigt wird. Für jeden zu produzierenden Ergebnistyp muss ein Projekttyp definiert werden. In diesem Beispiel ist das Ziel eine auf der ARM-Plattform ausführbare Datei. Projekttypelemente werden wie `tool` Elemente über das Kontextmenü der Erweiterung angelegt. Im ersten Schritt reichen wieder ID und lesbarer Name für die Konfiguration aus.

Für jede Projekttypdefinition muss mindestens eine *Konfiguration* erstellt werden, mit der dann z. B. eine Debug-Version mit integrierten Informationen für einen Debugger oder eine Produktionsversion, die diese Informationen nicht enthält, generiert werden kann. Eine neue Konfiguration wird über das Kontextmenü der Projekttypdefinition erstellt: *New* → *configuration*. Wieder muss eine Identifikationsnummer angegeben werden und es sollte ein verständlicher Name für die Konfiguration gewählt werden, z. B. *Produktion*. Im Feld `cleanCommand` wird der Befehl für das Entfernen der Ergebnisse im Erstellungsprozess angegeben

werden, z. B. `rm -f` für das Löschen von Dateien. Für die Zuordnung von Fehlermeldungen, die von den Werkzeugen dieser Konfiguration zurückgeliefert werden, zu der auslösenden Stelle im Quellcode, sind so genannte Fehlerparser (engl.: Error Parser) zuständig. Bereits vorhandene Fehlerparser sind z. B.

```
org.eclipse.cdt.core.MakeErrorParser
org.eclipse.cdt.core.GCCErrorParser
org.eclipse.cdt.core.GLDErrorParser
org.eclipse.cdt.core.GASErrorParser
org.eclipse.cdt.core.VCErrorParser
```

Mehrere Fehlerparser werden durch Semikolon getrennt. Sollen hier zusätzlich Fehlerparser angegeben werden, die nicht Teil der CDT sind, muss für das Plug-In ein Aktivator erstellt und im Manifest-Editor entsprechend konfiguriert werden.

Der nächste Schritt ist die Zusammenstellung einer *Toolchain*-Definition für die gerade definierte Projektkonfiguration. Ein neues `toolchain`-Element wird für die Konfiguration über das Kontextmenü des `configuration`-Elements hinzugefügt. Eine Konfiguration kann mehrere `toolchain`-Definitionen beinhalten. Da das Ausführen der Toolchain an die Plattform gebunden ist, für die sie erstellt wurde, sollte das neue Plug-In sinnvollerweise nur dann für die Erstellung neuer Projekte angeboten werden, wenn Eclipse auf dieser Plattform gestartet ist. Für eine Toolchain auf Linux, wie in diesem Fall, wird als Wert `linux` im Feld `osList` eingetragen. Damit können Eclipse-Projekte, die mit dieser Toolchain entwickelt werden, nur unter Linux angelegt und verwendet werden. Im Feld `targetTool` wird die Identifikation der `tool`-Definition angegeben, die das Endergebnis dieser Konfiguration erstellen kann, in diesem Fall also die ID des Linkers.

Jede Toolchain benötigt immer ein Erstellungswerkzeug (engl.: Builder), sowie mindestens ein Werkzeug, um Zwischen- und Endergebnisse des Erstellungsprozesses herzustellen. Innerhalb des `toolchain`-Elements werden entsprechende Elementdefinitionen angelegt, zunächst das Erstellungswerkzeug über das Element `builder`. Als Wert für das Feld `command` wird normalerweise der Befehl `make` angegeben.

Nun muss noch der oben definierte Übersetzer und Linker in die Toolchain integriert werden. Das funktioniert, indem innerhalb des Toolchain Elements zwei neue `tool` Elemente angelegt und als Wert für das Feld `superClass` die Identifikationen der vorher definierten Werkzeuge eingetragen wird. Über die Identifikationen werden die Werkzeugdefinitionen referenziert und somit in die Toolchain integriert.

Damit ist das Grundgerüst für Eclipse Entwicklungsprojekte für die neue Plattform einsatzbereit. Das neue Plug-In kann in einer Eclipse Testumgebung getestet werden. Dazu wird das Plug-In Entwicklungsprojekt in der Sicht *Package-Explorer* ausgewählt und dann in der Eclipse Menüzeile über *Run* → *Debug as* → *Eclipse Application* die Testumgebung gestartet. In dieser zweiten Eclipse Umgebung kann nun ein neues *C-Project* angelegt und als Projekttyp die neue Definition ausgewählt werden.

Die jetzt vorliegende Konfiguration verfügt noch über keine durch den Benutzer veränderbare Parameter, um den Erstellungsprozess zu beeinflussen. Für jedes Werkzeug werden in den CDT *Optionen* definiert, die wiederum in *Kategorien* logisch geordnet werden. Zunächst muss also in der Sicht für Plug-In Entwicklung eine Kategorie angelegt und dem Werkzeug zugeordnet werden. Das funktioniert über das Kontextmenü des Werkzeugs (`tool`) auf der Seite *Dependencies* des Manifest-Editors. Neben einer eindeutigen ID wird auch hier der Name der Kategorie in den Textfeldern `id` und `name` festgelegt. In diesem Beispiel erhält die Kategorie den Namen *Allgemein*.

Nun können Optionen für diese Kategorie definiert werden. Wie Optionskategorien werden auch Optionen über das Kontextmenü des Werkzeugs definiert. Zum Beispiel kann bei dem hier verwendeten Übersetzer mit der Option `-B` eine Liste mit Verzeichnissen angegeben werden, die nach Quelldateien durchsucht werden sollen. Neben einer eindeutigen ID und einem Namen muss noch der Parametertyp im Feld `valueType` ausgewählt werden, in diesem Fall `string`. Im Feld `command` wird der eigentliche Parameter `-B` angegeben und im Feld `category` muss die ID der oben definierten Optionenkategorie eingetragen werden. Der Wert für `browseType` ist `directory` und für `category` wird die Identifikation der eben erstellen Kategorie eingetragen.

In diesem Beispiel wurde die minimale Konfiguration für die Integration einer neuen Toolchain in die Eclipse CDT beschrieben. Die Erweiterungspunkte der CDT bieten darüberhinaus eine Fülle an Konfigurationsparametern und -möglichkeiten. Diese Flexibilität erlaubt es, praktisch jedes beliebige Werkzeug in die Entwicklungsumgebung zu integrieren.

5.4.1.4 Device Software Development Platform (DSDP)

Im Eclipse Projekt *DSDP* werden eine Reihe weiterer Unterprojekte zusammengefasst, die sich speziell mit der Entwicklung von Software für das Pervasive Computing beschäftigen.

Target Management

Im Projekt *Target Management* werden Datenmodelle und Schnittstellen entwickelt, um entfernte Systeme (engl.: Remote Systems) zu konfigurieren und zu verwalten. Entferntes System ist hier ein Überbegriff für Computer, auf die über ein TCP/IP Netzwerk zugegriffen werden kann. Diese Definition schliesst eingebettete Systeme unter Linux ein.

Die Vielzahl unterschiedlicher Systeme und Möglichkeiten, auf diese zuzugreifen, erschwert die Entwicklung für diese Systeme. Das Projekt Target Management möchte den Zugriff auf entfernte Systeme vereinheitlichen und somit vereinfachen. Es stellt eine erweiterbare Plattform bereit, die den Zugriff auf ein entferntes System hinter einer definierten Schnittstelle verbirgt.

Kernbestandteil des Projektes ist die Anwendung *Remote System Explorer* (*RSE*). Mit RSE wird auf die Zielumgebung zugegriffen und diese verwaltet. Zum

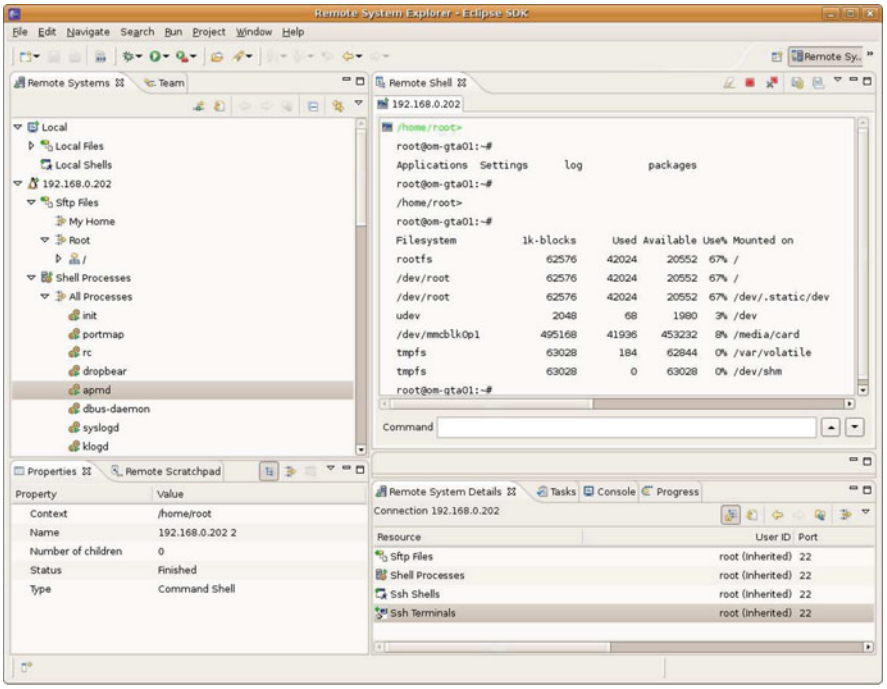

Abb. 5.7 Die Remote System Explorer Perspektive

Beispiel können Daten auf die Zielumgebung und von dieser übertragen oder die auf der Zielumgebung ausgeführten Prozesse überwacht werden. Dafür bietet das Target Management Projekt eine eigene Perspektive an und unterstützt bereits eine Reihe gängiger Verbindungsprotokolle und Zielsysteme, u.a. natürlich Linux. Ausserdem besteht die Möglichkeit, auf dem Zielsystem Kommandozeileninterpreter (engl.: Shell) zu öffnen und diese innerhalb von RSE zu verwenden. Abbildung 5.7 zeigt die RSE Perspektive mit geöffneter Shell und der Anzeige ausgeführter Prozesse auf dem Zielsystem. Um die Verbindung zu einem Zielsystem zu vereinfachen, bietet das Target Management Projekt einen Server-Dienst an. Dieses Dienstprogramm wird auf der Zielumgebung installiert und gestartet. Für die Kommunikation zwischen RSE und dem Dienst, wird das *Data Store* Protokoll verwendet.

Mobile Tools for Java (MTJ)

Als Unterprojekt von DSDP beschäftigt sich *MTJ* mit der Entwicklung von Programmen für die Java Plattform auf mobilen Geräten, also Softwareentwicklung für die Java Micro Edition (siehe Abschn. 6.2.2.1). Basis für die Entwicklung des MTJ Projektes ist *EclipseME* (http://eclipseme.org/), ein Eclipse Plug-In für die MIDlet-Entwicklung.

Tools for Mobile Linux (TmL)

Mit den *Tools for Mobile Linux* wird Eclipse um Werkzeuge zur Entwicklung von C und C++ Programmen für Pervasive Linux erweitert. Dabei soll der gesamte Entwicklungsprozess vom Entwurf über Entwicklung und Debugging bis zur Installation auf der Zielplattform abgedeckt werden.

5.4.1.5 Embedded Rich Client Platform (eRCP)

Das Project *eRCP* als Teil des Eclipse Runtime (RT) Projekts hat zum Ziel, die Eclipse Rich Client Platform (RCP) derart zu erweitern, dass RCP-basierte Programme auch auf eingebetteten Systemen eingesetzt werden können. Das hat den Vorteil, dass das gleiche Programmiermodell wie für RCP-Anwendungen für Arbeitsplatzrechner und Server verwendet wird. Damit erweitert sich die Portierbarkeit von RCP Programmen auf eingebettete Systeme.

5.4.1.6 Pulsar

Mit dem *Pulsar*-Projekt soll die Integration in Eclipse von unterschiedlichen Entwicklungswerkzeugen für die mobile Entwicklung vereinfacht werden. Pulsar for Mobile Developers besteht aus einer auf der Eclipse-Plattform aufbauenden Anwendung, die auf Knopfdruck die benötigten Entwicklungsumgebungen für mobile Plattformen diverser Hersteller installiert. Damit wird der Tatsache Rechnung getragen, dass die Software-Entwicklung für unterschiedliche Geräte im Umfeld des Pervasive Computing nach wie vor nicht problemlos mit einem einheitlichen Satz an Entwicklungswerkzeugen bewerkstelligt werden kann. Informationen über Pulsar sind unter http://www.eclipse.org/pulsar/ zu finden.

Ausserdem nutzt Pulsar den Bekanntheitsgrad von Eclipse und das verbreitete Bedienungskonzept. Momentan besteht Pulsar neben der Eclipse Platform als Basis aus den Java Development Tools (JDT), Mobile Tools for Java (MTJ), Sequoyah, Mylyn und dem Plugin Development Environment (PDE).

5.5 Entwicklung auf der Zielplattform

Falls die Möglichkeit besteht, eine Zielplattform mit den benötigten Prozessor- und Speicherressourcen auszustatten, kann die Entwicklung auch direkt auf der Zielplattform erfolgen. Das hat den Vorteil, dass kein Cross-Compiler eingesetzt werden muss.

Wenn die Rechenkapazität bezüglich Prozessorleistung und Arbeitsspeicher ausreicht und das Zielgerät über eine Netzwerkschnittstelle verfügt, lässt sich der benötigte nicht flüchtige Speicherplatz für die Installation der Entwicklungswerkzeuge sehr einfach mit dem Netzwerkprotokoll *Network File System* (*NFS*) in den Verzeichnisbaum einhängen. Auf diese Weise muss der Speicherplatz nicht direkt auf dem Zielgerät verfügbar sein, sondern wird über das Netzwerk bereitgestellt.

Eine andere Alternative ist es, die benötigten Programme auf ein externes Speichermedium wie eine CompactFlash Speicherkarte oder eine Festplatte zu kopieren und dieses in den Verzeichnisbaum des Zielgerätes einzuhängen, sofern dies von dem Gerät unterstützt wird.

Je leistungsfähiger die Umgebungen im Pervasive Computing werden, desto mehr am Entwicklungsprozess beteiligte Softwarekomponenten können direkt auf dem Gerät ausgeführt werden. Selbst integrierte grafische Entwicklungsumgebungen für den direkten Einsatz auf dem Gerät sind inzwischen verfügbar.

In beiden Fällen muss eine auf der Zielplattform lauffähige Entwicklungsumgebung vorliegen. Das folgende Beispiel zeigt, wie eine GCC Toolchain mit Hilfe des in Abschn. 5.1.1 erstellten Cross-Compilers für die ARM Plattform erstellt wird.

Für das Beispiel wird ein Speichermedium mit ausreichender Kapazität auf dem Entwicklungsrechner unter /mnt/md in den Verzeichnisbaum eingehängt. Dabei ist wichtig, dass das verwendete Dateisystem Verweise (engl.: Links), insbesondere „Soft Links“ unterstützt und dass die Benutzerkennung, mit der die Toolchain erstellt wird, Schreibrecht auf dieses Verzeichnis hat.

Im ersten Schritt werden das Werkzeug `make` sowie die *binutils* mit einem Cross-Compiler für die Zielplattform übersetzt. `make` kann ebenso wie die Bestandteile der GNU GCC Toolchain von der Webseite http://www.gnu.org heruntergeladen werden. Der Quellcode wird in ein temporäres Verzeichnis entpackt und korrespondierende Verzeichnisse für die Übersetzungsprodukte anglegt. Damit für den Übersetzungsvorgang der Cross-Compiler verwendet wird (in diesem Fall `arm-linux-gcc`) muss sich dieser im Suchpfad (`$PATH`) für ausführbare Dateien befinden. Das Konfigurationsskript wertet die Umgebungsvariable `CC` aus und setzt den darin definierten Übersetzer in die erzeugten Datei `Makefile` ein. Daher werden diese Variablen vor dem Aufruf von `configure` entsprechend gesetzt.

Mit den Konfigurationsparametern `--host` und `--prefix` wird dem Skript wie in Abschn. 5.1.1 beschrieben mitgeteilt, für welche Zielplattform übersetzt und welches Verzeichnis für die Installation verwendet werden soll. Für die *binutils* ist zusätzlich die Angabe der Rechnerarchitektur, auf dem die Toolchain erstellt wird, in Form von `--build=i386` notwendig.

```
export PATH=/usr/local/arm/bin:$PATH
export CC=/usr/local/arm/bin/arm-linux-gcc
gunzip make-3.80.tar.gz
tar -xvf make-3.80.tar
mkdir make
cd make
../make-3.80/configure --prefix=/mnt/md --host=arm-linux
make
make install

cd ..
gunzip binutils-2.14.tar.gz
tar -xvf binutils-2.14.tar
mkdir binutils
```

```
cd ../binutils
../binutils-2.14/configure --prefix=/mnt/md --host=arm-linux \
 --build=i386
make
make install
```

Anschliessend werden die für die Erstellung des GCC Übersetzers die Kernel-Header Dateien der Zielumgebung in das Verzeichnis der `binutils` kopiert. Diese Header Dateien stammen aus den konfigurierten Quellen des auf der Zielplattform eingesetzter Kernels. Ausserdem werden die GCC Quellen entpackt.

```
cp -dR tmp/hhkernel/linux/kernel/include/asm-arm \
 /mnt/md/include/asm
cp -dR tmp/hhkernel/linux/kernel/include/linux \
 /mnt/md/include/linux

gunzip gcc-4.0.2.tar.gz
tar -xvf ../gcc-4.0.2.tar
```

Vor dem Aufruf des Konfigurationsskripts `configure` muss noch eine Anpassung in einer Datei aus dem GCC Quellcode vorgenommen werden. In der Datei `gcc-4.0.2/gcc/config/host-linux.c` wird die nicht mehr deklarierte Makrodefinition `SSIZE_MAX` verwendet. Daher ist es nötig, sie in Zeile

```
      nbytes = read (fd, base, MIN (size, SSIZE_MAX));
```

entsprechend zu ersetzen:

```
      nbytes = read (fd, base, MIN (size,
 INTTYPE_MAXIMUM(ssize_t)));
```

Danach wird eine auf der Zielplattform ausfürbare Version des GCC Übersetzers konfiguriert, übersetzt und installiert.

```
mkdir gcc
cd gcc
../gcc-4.0.2/configure --prefix=/mnt/md --host=arm-linux \
 --target=arm-linux --build=i386 --with-cpu=strongarm110
make
make install
```

Auf die gleiche Weise wie GCC muss noch die Sprachbibliothek Glibc mit vollem Sprachumfang für die Zielplattform übersetzt werden. Dazu wird die Sprachbibliothek Glibc entpackt, konfiguriert, übersetzt und installiert.

```
gunzip glibc-2.3.6.tar.gz
tar -xvf glibc-2.3.6.tar
mkdir glibc
cd glibc
```

```
../glibc-2.3.6/configure --prefix=/mnt/md --host=arm-linux \
 --target=arm-linux --build=i386 --enable-add-ons=linuxthreads
make
make install
```

Von hier an kann mit der Toolchain direkt auf der Zielplattform entwickelt werden. Die Ergebnisse der bisherigen Schritte werden in den Verzeichnisbaum der Zielumgebung eingebunden und die Umgebungsvariable `PATH` so gesetzt, dass die ausführbaren Programme der Toolchain vom System gefunden werden.

Die GCC steuern über eine Konfigurationsdatei `specs`, wo sich die zu verwendenden Werkzeuge der *binutils* wie z. B. der Linker befinden. Seit Version 4 der GCC wird diese Datei nicht mehr separat installiert, sondern es wird eine in die GCC integrierte Variante verwendet. Da sich auf dem Entwicklungsrechner, auf dem die Toolchain übersetzt wurde, die *binutils* in einem anderen Verzeichnis befinden, als in der Zielumgebung, muss die Datei `specs` entsprechend angepasst werden, damit es nicht zu Fehlern beim Linken kommt, da die GCC den Linker nicht finden kann. Falls eine Datei `specs` an der von der GCC erwarteten Stelle im Dateisystem der Zielumgebung existiert, wird deren Inhalt ausgelesen und nicht die integrierte Version verwendet. Welche Version der `specs` Datei von der GCC aktuell verwendet wird, kann mit dem Parameter `-v` ermittelt werden.

Mit Hilfe des Parameters `-dumpspecs` gibt die GCC den Inhalt der integrierten `specs` auf die Konsole aus. Die Ausgabe wird in eine `specs` Datei umgeleitet und kann nun bearbeitet werden.

```
gcc -dumpspecs > /media/cf/lib/gcc/arm-linux/4.0.2/specs
```

Da die Header Dateien der Sprachbibliothek nicht in das Standardverzeichnis `/usr/include` installiert wurden, wird die Umgebungsvariable `CPPFLAGS` mit einem Kommandoparameter für den C Präprozessor belegt. Mittels des Parameters `-I/media/cf/include` wird der Präprozessor mitgeteilt, wo sich die Header Dateien befinden.

```
mount -text3 /dev/hda1 /mnt/cf
export PATH=$PATH:/mnt/cf/bin
export CPPFLAGS=-I/mnt/cf/include
```

5.6 Paketierung und Softwareverwaltung

Zu den Aufgaben bei der Verwaltung von Software in einem Computersystem gehört u.a. die Installation und das Entfernen von Software, die Versionsverwaltung und die Prüfung von Abhängigkeiten zu anderen Softwarebestandteilen im System. Zur Unterstützung und Erleichterung dieser Tätigkeiten haben sich im Arbeitsplatzumfeld für Linux die Werkzeuge RPM und APT etabliert. RPM (RPM Package Mangager) wurde wie der Name andeutet von der Firma Red Hat entwickelt und

APT (A Package Tool) ging aus dem Debian-Projekt hervor. Beide Werkzeuge ermöglichen umfangreiche Konfigurationsmassnahmen zur Softwareverwaltung.

Der mächtige und komplexe Funktionsumfang beider Werkzeuge zieht allerdings einen beträchtlichen Ressourcenbedarf nach sich und macht sie daher für den Einsatz im Pervasive Computing weitgehend unbrauchbar. Allerdings gibt es auch für einige Plattformen Hilfsmittel, deren Konzepte teilweise von den genannten Werkzeugen übernommen wurden, bei deutlich verringertem Ressourcenbedarf.

5.6.1 Itsy Package Management System (iPKG)

Das Itsy Package Management System wird im Rahmen des Familiar-Projektes entwickelt und ist auch Bestandteil der Familiar-Distribution. Die Funktionalität wurde von Debian APT übernommen und auf die wichtigsten Funktionen beschränkt, z. B. wurde aus Platzgründen auf Dokumentation innerhalb eines Paketes verzichtet.

Mit iPKG lassen sich installierte Systeme komfortabel auf einen definierten Software-Versionsstand bringen.

Das Projekt Openmoko hat die Konzepte von iPKG übernommen und betreibt unter dem Namen *opkg* die Weiterentwicklung.

5.6.1.1 IPK Dateiformat

Das iPKG Werkzeug `ipkg` arbeitet mit Dateien der Dateiendung `.ipk`. Diese Dateien sind Installationspakete und beinhalten sowohl die zu installierenden Programmdaten als auch Metainformation, die von ipkg für die Installation und Konfiguration ausgewertet werden.

Die Paketdatei selbst ist ein Dateiarchiv, das mit dem Archivierungswerkzeug `ar` erstellt wird. `ipkg` erwartet in diesem Archiv folgende Dateien:

data.tar.gz
: Eine komprimierte Archivdatei mit den Programmdateien der zu installierenden Software. Dabei muss die Verzeichnisstruktur genau so aufgebaut sein, wie sie nach der Installation auf dem Zielgerät vorliegen soll.

control.tar.gz
: Ebenfalls eine komprimierte Archivdatei. Dieses Archiv enthält ein Unterverzeichnis `CONTROL`, in der obersten Verzeichnisebene. Dieses Verzeichnis muss eine Datei `control` beinhalten mit den Metainformationen die den Installationsprozess steuern. Der Inhalt dieser Datei wird weiter unten beschrieben. Die übrigen Dateien im Verzeichnis `CONTROL` sind optional.
Falls das Paket Konfigurationsdateien enthält, die im Fall eines Software-Update beibehalten und nicht überschrieben werden sollen, muss eine zusätzliche Datei `conffiles` angelegt werden. Diese Datei enthält für jede Konfigurationsdatei eine Zeile mit dem absoluten Dateipfad der entsprechenden Datei.

Vor und nach der Installation bzw. Deinstallation können Skripte auf der Zielplattform aufgeführt werden, um Konfigurationsschritte durchzuführen. Die Namen der Skriptdateien, die vor bzw. nach einer Installation ausgeführt werden, sind `preinst` und `postinst`, die Gegenstücke für eine Deinstallation heissen entsprechend `prerm` und `postrm`. Diese Skripte müssen nach erfolgreicher Ausführung den Rückgabewert `0` zurückliefern, ansonsten wird das Softwarepaket nicht installiert. Innerhalb der Skripte kann über die Umgebungsvariable `PKG_ROOT` auf den Inhalt des Paketes verwiesen werden. Ausserdem können über die Standardeingabe Informationen vom Benutzer erfragt werden.

debian-binary — Diese Datei wird momentan von `ipkg` ignoriert, muss aber vorhanden sein.

Für die Anwendung des Bluetooth Applets der OPIE Oberfläche beispielsweise enthält die Datei `control.tar.gz` die folgende Dateien:

```
/opt/QtPalmtop/pics/bluetoothapplet/bluezon.png
/opt/QtPalmtop/pics/bluetoothapplet/bluezoff.png
/opt/QtPalmtop/plugins/applets/libbluetoothapplet.so
```

Die bereits erwähnte Steuerungsdatei `CONTROL/control` mit Metainformationen für die Installation besteht aus paarweisen Einträgen von Schlüsselworten und den zugehörigen Werten im Format `<Schlüsselwort>: <Wert>`, z. B. `Version: 1.0`. Die folgenden Schlüssel-Wertpaare müssen immer angegeben werden:

Package — Der Name des zu installierenden Softwarepakets.

Version — Die Versionsinformation des Pakets. Teil der Versionsinformation sollte mindestens eine Ziffer sein, die bei neuere Versionen hochgezählt wird.

Architecture — Die Hardwarearchitektur der Zielplattform für die die Software dieses Pakets übersetzt ist. Gültige Einträge sind momentan `arm` und `all`.

Maintainer — Die Kontaktdaten der für die Pflege der Software verantwortlichen Person oder Organisation. Üblicherweise werden hier Namen und E-Mailadressen angegeben.

Section — Kategorisierungsinformation, um welche Art Software es sich handelt, z. B. `admin`, `base`, `comm`, `editors`, `extras`, `graphics`, `libs`, `misc`, `net`, `text`, `web`, `x11`.

Priority — Dieser Eintrag spezifiziert die Relevanz des Pakets für das Gesamtsystem, gültige Werte sind `required`, `standard`, `important`, `optional`, `extra`. Anwendungsprogramme sollten normalerweise als `optional` gekennzeichnet werden.

Description Eine kurze Beschreibung des Softwarepakets. Falls sich die Beschreibung über mehrere Zeilen erstreckt muss jede neue Zeile mit einem Leerzeichen eingerückt werden.

Zusätzlich zu den zwingend erforderlichen Angaben, gibt es optionale Einträge.

Depends Falls die Funktionsfähigkeit der Software des Pakets von anderen Paketen abhängt, kann das mit Hilfe des zusätzlichen Felds `Depends` spezifiziert werden. Als Wert für dieses Schlüsselwort, wird in diesem Fall die Liste der abhängigen Paketnamen, getrennt durch Kommata angegeben. Ist die Software von einer bestimmten Version eines Paketes abhängig, wird die Versionsnummer in runden Klammern an den Paketnamen angehängt. Um die Abhängigkeit von einer exakten Versionsnummer zu minimieren, sollte ein Wertebereich angegeben werden, z. B. >= 1.2.0.

Source Mit Hilfe dieses Schlüsselwortes wird erklärt, wo die Codequellen des Softwarepaketes zu finden sind.

Die Datei `CONTROL/control` des oben genannten Bluetooth-Pakets beispielsweise hat den Inhalt:

```
Package: opie-bluetoothapplet
Version: 1.2.0-r0
Description: Opie Bluetooth Applet
Section: opie/applets
Priority: optional
Maintainer: Team Opie <opie@handhelds.org>
Architecture: arm
OE: opie-bluetoothapplet-1.2.0
Depends: libqpe1 (>= 1.2.0), libopietooth1-1 (>= 1.2.0), libqte2
(>= 2.3.10), libgcc1 (>= 3.4.3), libc6 (>= 2.3.2+cvs20040726)
Source: cvs://anoncvs:anoncvs@cvs.handhelds.org/cvs;tag=v1_2_0;mo
dule=opie/noncore/net/opietooth/applet cvs://anoncvs:anoncvs@cvs.
handhelds.org/cvs;tag=v1_2_0;module=opie/pics/bluetoothapplet
```

Die Erstellung von `ipk` Dateien wird durch Werkzeugskripte vereinfacht, die unter ftp://ftp.handhelds.org/packages/ipkg-utils verfügbar sind.

Um eine `ipk` Datei zu erstellen, wird ein Verzeichnis mit dem Namen der zu erstellenden `ipk` Datei angelegt und darin die Verzeichnis- und Dateistruktur erstellt, wie sie auf der Zielplattform aufgebaut werden soll. Das Unterverzeichnis `CONTROL` wird ebenfalls auf oberster Ebene in diesem Unterverzeichnis erstellt. In dieses Verzeichnis werden die Dateien `control` sowie die oben beschriebenen optionalen Konfigurations- und Skriptdateien kopiert. Die ebenfalls benötigte Datei `debian-binary` wird von dem Skript erstellt.

Das Skript `ipkg-build` erzeugt eine `ipk` Datei mit dem Namen des Unterverzeichnisses, in dem sich die Paketdaten befinden:

```
ipkg-build [-c] [-C] [-o owner] [-g group] <pkg_directory> \
 [<destination_directory>]
```

Die Parameter haben folgende Bedeutung:

-c	Die `ipk` Datei wird als komprimiertes Archiv (`tar.gz`) und nicht mit dem Werkzeug `ar` erstellt.
-C	Ohne diesen Parameter löscht das Skript Dateien, die mit einer Tilde (`~`) enden. Mit diesem Parameter wird eine Warnung ausgegeben.
-o	Falls das Skript nicht als `root` ausgeführt wird, wird der Eigentümer der Dateien mit `-o root` als `root` definiert.
-g	Wie der Eigentümer der Dateien kann auch die Gruppe gesetzt werden.
pkg_directory	Mit diesem Parameter wird das Verzeichnis angegeben, in dem sich die Dateien für die zu erstellende `ipk` Datei befinden.
destination_directory	Das Verzeichnis, in das die erstellte `ipk` Datei abgelegt werden soll.

Neben der vereinfachten Erzeugung von `ipk` Dateien haben die Skripte den Vorteil, dass sie eine Konsistenzprüfungen mit den verarbeiteten Dateien durchführen. Damit können z. B. Syntaxfehler in den Konfigurationsdateien frühzeitig erkannt und korrigiert werden.

Ausser dem hier vorgestellten zentralen Skript `ipk-build` enthält die Zusammenstellung weitere nützlich Skripts zur Bearbeitung von `ipk` Dateien.

5.6.1.2 Der ipkg Befehl

Das Programm `ipkg` dient dazu, Software auf der Zielplattform zu zu installieren, zu konfigurieren, aktualisieren und zu entfernen:

```
ipkg [Optionen...] Befehl [Argumente...]
```

Die von ipkg akzeptierten Befehle sind:

update	Aktualisiere die Liste verfügbarer Pakete.
upgrade	Bringe alle installierten Pakete auf den neuesten Versionsstand.
install <*pkg*>	Lade das Paket < *pkg* > und alle Abhängigkeiten herunter und installiere sie.
install <*datei.ipk*>	Installiere das Paket <*datei.ipk*>.
configure	Konfiguriere entpackte Pakete. Optional können bestimmte Pakete angegeben werden.
remove <*pkg*|*regexp*>	Entferne Paket <*pkg*> oder alle Pakete, die durch den regulären Ausdruck <*regexp*> definiert werden.
flag <*flag*> <*pkg*> ...	Kennzeichne ein oder mehrere Pakete mit einem der Werte `hold`, `noprune`, `user`, `ok`, `installed`, `unpacked`. Pro Aufruf kann ein Wert gesetzt werden.

5.6.1.3 Softwareverteilung und Versionsverwaltung

Das Konzept der Softwareverwaltung mit Hilfe von `ipkg` sieht u.a. die Installation von Software von einem zentralen Rechner vor, auf dem die jeweils aktuelle Version der Software in Form von `ipk` Dateien bereitgestellt wird. Das ist eine entscheidende Funktionalität, wenn es darum geht, eine Anzahl eingesetzter Endgeräte mit aktuellen Softwareversionen zu versorgen. Die Versionsstände der zu verwendenden Software werden zentral auf einem Server gepflegt und die Zielumgebungen laden sich in regelmässigen Zeitabständen die jeweils aktuelle Version herunter und installieren diese.

Dies vereinfacht die Softwareverteilung und Versionsverwaltung deutlich, der Nutzen wächst mit der Anzahl eingesetzter Geräte. Die einzige Voraussetzung für diese Art der Versionsverwaltung ist ein Netzwerkzugang der Endgeräte.

Die Linux-Distribution Familiar verwendet für die Verwaltung der in der Distribution enthaltenen Softwarepakete den von `ipkg` vorgesehenen Mechanismus zur Installation von Software-Updates. Es ist also möglich, die Bestandteile der Distribution über das Internet auf dem aktuellen Stand zu halten.

Ergänzend dazu können mehrere eigene Server konfiguriert werden, die z. B. in einem Firmennetzwerk diese Software über eine schnelle Netzwerkverbindung bereitstellen oder auch selbst entwickelte Software auf diesem Weg verteilen. Eine solche Zusammenstellungen von Softwarepaketen wird *Feed* genannt, weil auf diese Weise die Software sozusagen in den Verteilungskanal eingespeist wird.

Auf einem eigenen Feed-Server werden die zu verteilenden Softwarepakete in Form von `ipk` Dateien in einem Verzeichnis zusammengestellt und mit Hilfe der bereits erwähnten `ipkg`-Werkzeuge ein Index generiert, der von den Endgeräten abgefragt werden kann. Diese Indexdatei muss den Dateinamen Packages haben und wird unter Linux oder UNIX mit folgendem Befehl erzeugt, wobei `<paketverzeichnis>` für das Verzeichnis steht, in dem sich die `ipk` Dateien befinden:

```
ipkg-make-index <paketverzeichnis> > <paketverzeichnis>/Packages
```

Das Paketverzeichnis muss nun noch über FTP oder HTTP erreichbar gemacht werden. Damit ist der Feed-Server einsatzbereit.

Als Einschränkung gilt, dass momentan nur genau eine Version jedes Softwarepaketes im Feed-Verzeichnis vorhanden sein darf, da `ipkg` sonst nicht fehlerfrei funktioniert.

5.6.2 Open Services Gateway Initiative (OSGi)

Die *Open Services Gateway Initiative* ist ein offener, schlanker und sicherer Industriestandard zur Verwaltung von Softwarekomponenten, so genannten *Bundles* für die Java Plattform (Abschn. 6.2.2) auf vernetzten Geräten. Ursprünglich wurde der Standard für den Betrieb auf Internet-Zugangsgeräten zur Automatisierung in

Haushalten entwickelt, war also schon immer für den Einsatz im Pervasive Bereich gedacht. Das Eclipse-Projekt *Equinox* (Abschn. 5.4) stellt die Referenzimplementierung des Standards in der Version R4 dar.

OSGi definiert Dienste (engl.: Services) in Form von Java-Schnittstellen, die von Bundles implementiert werden können. Beispielsweise gibt es Dienste, um Bundles zu installieren und zu entfernen, zu starten und zu stoppen, ohne die Notwendigkeit eines Neustarts des gesamten Geräts bzw. Betriebssystems. Ausserdem können Abhängigkeiten zwischen Bundles definiert werden. Vorangetrieben wird die Standardisierung von der *OSGi Alliance* einem nicht gewinnorientierten Zusammenschluss zahlreicher Unternehmen. Auf der Webseite http://www.osgi.org veröffentlicht die OSGi Alliance den Standard und informiert über aktuelle Entwicklungen.

Dienste werden in einem zentralen Diensteverzeichnis (engl.: Service Registry) registriert. Über dieses Verzeichnis können Dienste gefunden und von anderen Diensten verwendet werden. Ausserdem ist es möglich, auf die Verfügbarkeit eines bestimmten Dienstes zu warten und beim Start des gewünschten Dienstes benachrichtigt zu werden. Abbildung 5.8 zeigt vereinfacht das Konzept der OSGi-Architektur.

Durch dieses Konzept wird eine lose Kopplung zwischen Bundles erreicht, was wiederum zu hoher Stabilität und grosser Flexibilität bei verringerter Komplexität

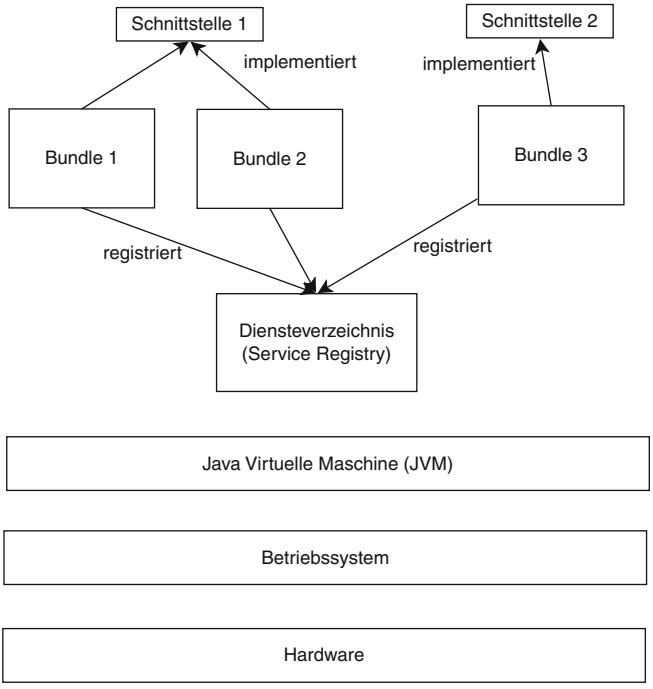

Abb. 5.8 Die OSGi Architektur

führt. Bundles sind typischerweise kleine Funktionalitätseinheiten und können zur Laufzeit ausgetauscht werden. Aufgrund der leichtgewichtigen Architektur eignet sich OSGi sehr gut für den Einsatz im Pervasive Computing. Daher wird OSGi inzwischen in unterschiedlichsten Bereichen eingesetzt, in Mobiltelefonen, im Fahrzeugbereich, auf Standard-PCs und im Serverumfeld.

Sicherheit innerhalb der Plattform wird auf verschiedene Weise erreicht. Zunächst bietet die Java-Plattform viele Programmierkonstrukte gar nicht an, die von bösartiger Software missbraucht werden könnten, um im System Schaden anzurichten. Beispielsweise ist direkter Zugriff auf Speicheradressen nicht möglich. Darüber hinaus stellt die Java-Plattform ein Konzept der Zugriffsberechtigungen bereit, mit dessen Hilfe mehrere Programme innerhalb der selben virtuellen Maschine ohne gegenseitigen Zugriff ausgeführt werden können. Ausserdem sind die Bundles strikt voneinander getrennt und können nur über entsprechende Berechtigungen ihre Dienste gegenseitig nutzen.

5.6.3 Debian Package (dpkg)

Das Paketformat Debian Package (dpkg) stammt wie der Name schon sagt, von der Linux-Distribution Debian. Dieses im Desktop-Umfeld häufige Paketierungsformat mit der Dateiendung .deb kommt im Pervasive Linux Bereich eher selten vor. Allerdings stellt Busybox (Abschn. 3.6.2) die Befehle zum Umgang mit diesem Format zur Verfügung.

Kapitel 6
Anwendungs- und Systementwicklung

Softwareentwicklung für Pervasive Linux unterscheidet sich in manchen Bereichen von der Entwicklung für Desktop-Systeme, die grundsätzliche Vorgehensweise ist aber weitgehend identisch.

Abhängig von der Art der zu entwickelnden Anwendung müssen entsprechende Werkzeuge und Bibliotheken ausgewählt werden. In diesem Kapitel wird die Vorgehensweise für eine Reihe von Umgebungen beschrieben. Eine Umgebung stellt in diesem Zusammenhang Schnittstellen für die Interaktion mit dem Benutzer bereit und legt u.U. die einzusetzende Programmiersprache fest.

6.1 Strukturierung

Um die Komplexität bei der Entwicklung zu reduzieren und den Ressourcenverbrauch im laufenden System zu optimieren, wird Software in logische Einheiten zerlegt, in denen zusammenhängende Funktionalität gruppiert wird. Auf der Ebene des Quellcodes kann die Grösse einer logischen Einheit bis auf Dateiebene reichen. In diesem Fall wird jede logische Einheit in einer eigenen Datei gespeichert. Beispielsweise ist es in der Programmiersprache Java üblich, jede Klasse in einer eigenen Datei zu verwalten (mit der Ausnahme von inneren Klassen).

Zwischen den Einheiten wird über definierte *Schnittstellen* kommuniziert. Alle modernen Programmiersprachen stellen eine Möglichkeit bereit Schnittstellen zu definieren.

6.2 Programmiersprachen

Die grosse Auswahl an Programmiersprachen für Linux reduziert sich im Pervasive Computing, von Spezialfällen abgesehen, meistens auf die hier vorgestellten. Eine Kombination aus Bekanntheitsgrad und Popularität bei Entwicklern, Verfügbarkeit der Sprachwerkzeuge sowie Portabilität und Optimierbarkeit des in der jeweiligen Sprache entwickelten Quellcodes sorgen für momentane Verbreitung der vorgestellten Programmiersprachen.

C. Czernohous, *Pervasive Linux*, X.systems.press,
DOI 10.1007/978-3-540-68426-8_6, © Springer-Verlag Berlin Heidelberg 2012

6.2.1 C

C ist die wichtigste Sprache für die Entwicklung unter Linux. Der Linux Kernel selbst ist in C geschrieben und die meisten Programme und Gerätetreiber für Linux ebenfalls. C wurde von Brian W. Kernighan und Dennis M. Ritchie ursprünglich für UNIX Systeme als prozedurale Sprache für allgemeine Einsatzzwecke entwickelt, dennoch ist C als Sprache unabhängig von Hardware-Umgebungen und Betriebssystemen (siehe [36]). Diese Voraussetzung ermöglicht erst die Portierung von Linux auf die zahlreichen heute unterstützten Hardware- und Software-Plattformen. Eine Übersicht über die Entstehungsgeschichte von C findet sich auf http://cm.bell-labs.com/cm/cs/who/dmr/chist.html.

Zwar ist die Sprache einerseits plattformunabhängig entworfen, andererseits existiert für jede unterstützte Plattform eine Implementierung mit individuellen Anpassungen und ist deshalb als relativ systemnahe Sprache einzustufen. So wird z. B. die Grösse (und damit der Speicherbedarf und Wertebereich) für primitive Datentypen von der Implementierung des Übersetzers festgelegt. Dies ist insbesondere für das Pervasive Computing wichtig, da hier eine Vielzahl unterschiedlicher Plattformen zum Einsatz kommt und zwischen den Implementierungen grosse Unterschiede bestehen.

Die Plattformunabhängigkeit von C bezieht sich daher auf die Sprache selbst und nicht auf mögliche Zwischenergebnisse des Übersetzungsprozesses. Das bedeutet, dass ein Programm in C zwar plattformunabhängig geschrieben werden kann, für jede Plattform aber erneut übersetzt werden muss. C Quellcode kann in unterschiedlichem Masse auf die Übersetzung für unterschiedliche Plattformen vorbereitet sein. Einerseits können in C Programme sehr maschinennah und damit praktisch nur schwer auf andere Plattformen portierbar entwickelt werden. Andererseits stehen zahlreiche Konstrukte zur Verfügung, um auf die Unterschiede möglicher Zielplattformen schon bei der Entwicklung eingehen zu können. Daraus ergibt sich die grosse Varianz an Portabilität der existierenden Software für Linux.

C wurde vom American National Standards Institute (ANSI) als Standard normiert und als ANSI C bzw. C89 veröffentlicht, um die Portabilität von in C geschriebenen Programmen zu erleichtern. Dieser Standard wiederum wurde von der International Organization for Standardization (ISO) übernommen und ist als C90 bekannt. Eine überarbeitete Spezifikation wurde von der ISO erneut normiert und als C99 danach auch vom ANSI übernommen.

Aufgrund der Hardware-Nähe lassen sich mit C sehr effiziente und ressourcenschonende Programme schreiben.

6.2.1.1 Schnittstellen

In der Programmiersprache C muss eine Funktion vor ihrer ersten Verwendung deklariert werden. Das heisst, dass der Funktionsname, die Typen der Übergabeparameter sowie der Typ des Rückgabewertes bekannt gemacht werden müssen. Um diese Funktionsdeklaration in möglichst wiederverwendbarer Form festzuhalten, gibt es das Konzept der so genannten *Header-Dateien*. Eine Header-Datei

deklariert die Funktionsköpfe, also Funktionsnamen sowie die Typen der Ein- und Rückgabewerte. Header-Dateien haben die Dateiendung `.h`. Folgende Datei `durchschnitt.h` deklariert eine Funktion, die zwei Eingabewerte vom Typ `float` erwartet und den Durchschnitt der beiden Werte in Form eines `float` zurückliefert.

```
float durchschnitt(float, float);
```

Die Implementierung, also das Stück des Quelltextes, das die Programmlogik enthält, wird Funktionsdefinition genannt. Das entsprechende Stück Code für die Implementierung in der Datei `durchschnitt.c` sieht folgendermassen aus:

```
#include "durchschnitt.h"

float durchschnitt(float liter, float kilometer)
{
  return liter / kilometer;
}
```

In der ersten Zeile wird der Funktionskopf über die `#include` Direktive des C-Präprozessors in die Datei eingefügt. Dadurch wird die Funktionsdeklaration durch den Präprozessor vor dem Übersetzungsschritt physisch in den Quellcode eingebunden, d.h. der Inhalt der entsprechenden Datei wird an der Stelle, an der die Direktive steht, eingefügt und dann dem Übersetzer übergeben. An einer folgenden Stelle im Quellcode erfolgt darauf die Definition und Implementierung. Das Einbinden in den Quellcode erfolgt in Quellcodedateien, in welchen die Schnittstelle verwendet wird, auf die gleiche Weise, z. B. in der Datei `verbrauch.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include "durchschnitt.h"

int main(int argc, char *argv[])
{
  printf("Verbrauch pro 100 Kilometer: %f\n",
    durchschnitt(atoi(argv[1]), atoi(argv[2])) * 100);
  return 0;
}
```

Die `#include` Direktive mit spitzen Klammern lässt den Präprozessor in den Standardverzeichnissen für Header-Dateien des Compilers nach der Datei suchen. Steht die Header-Datei in Anführungszeichen, sucht der Präprozessor zunächst relativ zu dem Verzeichnis, in dem sich die einbindende Datei befindet, bevor er die Standardverzeichnisse durchsucht.

6.2.1.2 Der Präprozessor

Die Programmiersprache ermöglicht vor dem eigentlichen Übersetzungsvorgang einen Verarbeitungsvorgang, der von dem so genannten *Präprozessor* durchgeführt wird. Anweisungen an den Präprozessor beginnen mit dem Sonderzeichen #. Der Präprozessor führt Anpassungen auf der Quellcode-Ebene durch. Dabei können andere Dateien an der Stelle des Aufrufs in den Quellcode eingefügt, Makros für

Textersetzung definiert und Anweisungen über die Auswertung von Bedingungen gesteuert werden.

Gerade für die Entwicklung für unterschiedliche Plattformen spielt der Präprozessor eine wichtige Rolle. Durch die Definition entsprechender Makros und das Einfügen plattformabhängiger Header-Dateien abhängig von bestimmten Konfigurationsvariablen, kann der Quellcode vor dem Übersetzen an die Besonderheiten der Zielplattform angepasst werden.

Der Präprozessor selbst ist unabhängig von der Programmiersprache C und kann auch für die Verarbeitung beliebiger Textdateien verwendet werden.

Einfügen von Dateien

In diesem Fall wird der Inhalt der einzufügenden Datei an die Stelle der Präprozessoranweisung eingefügt, als ob dieser Teil des Textes sei. Häufigster Anwendungsfall für diese Anweisung ist das Einfügen von Header-Dateien. Die dafür definierte Anweisung lautet `#define`.

```
#include <stdio.h>
```

Makros

Makros ermöglichen das Ersetzen von entsprechend als Makro gekennzeichneten Textbausteinen durch definierten Text. Mit der Anweisung `#define` wird ein Makro angelegt und mittels `#undef` wieder entfernt.

```
#define PLATFORM arm
```

Für die Bezeichner von Makros gelten dieselben Regeln wie für Variablenbezeichner.

Bedingungen

Mit Hilfe von Bedingungen kann die Anweisungsfolge des Präprozessors gesteuert werden. Die auf die Bedingungsanweisung `#if` folgenden Anweisungen werden bis zum nächsten `#endif`, `#elif` oder `#else` ausgeführt, wenn der auf die Bedingung folgende Ausdruck einen numerischen Integer Wert ungleich 0 ergibt.

Beispielsweise können abhängig von dem Wert einer Umgebungsvariablen bestimmte Präprozessoranweisungen ausgeführt werden.

```
#if LANG == de_DE
  #define LANGUAGE Deutsch
#endif
```

Bei `#if` Bedingungen kann mit dem Ausdruck `defined(Makro)` auf die Existenz eines Makros geprüft werden, alternativ dazu mit der Kurzform `#ifdef`. Die beiden folgenden Beispiele sind also austauschbar:

```
#if defined(LANG)
  #define MULTI
#endif

#ifdef LANG
  #define MULTI
#endif
```

Mit `#ifndef` kann das Fehlen eines Makros überprüft werden.

6.2.1.3 Programmbibliotheken

Programmbibliotheken sind Dateien, die übersetzten Quellcode enthalten, der von anderen Programmen verwendet werden kann. Für Funktionalität, die in identischer Weise von verschiedenen Programmen benutzt werden kann, bietet es sich an, diese in Form von Programmbibliotheken bereitzustellen.

Statische Bibliotheken

Statische Bibliotheken (engl.: *Static Libraries*) bestehen aus Objektcode, der in ein zu erstellendes Programm eingebaut wird. Per Konventionen haben statische Programmbibliotheken unter Linux die Dateiendung `.a`. Es handelt sich dabei um Archivdateien, die mit dem Werkzeug `ar` erstellt werden.

Statische Bibliotheken sind heutzutage nur noch für spezielle Anwendungsbereiche interessant. Ursprüngliche Vorteile wie kürzere Übersetzungszeit und schnellere Ausführungszeit spielen heutzutage meistens keine Rolle mehr.

Gemeinsam genutzte Bibliotheken

Gemeinsam genutzte Programmbibliotheken (engl.: *Shared Libraries*) werden nicht fest in eine Programmdatei integriert, sondern erst zur Laufzeit eines Programms in den Speicher geladen und verwendet. Das hat den Vorteil, dass der Programmcode einer Bibliothek nur einmal im Speicher vorliegen muss und sich mehrere Programme den selben Speicherplatz teilen können.

Dateinamen für gemeinsam genutzte Bibliotheken unter Linux folgen einer Namenskonvention. Für jede Bibliothek muss eine Datei mit dem *Linker Name* existieren. Der Linker Name besteht aus der voll qualifizierten Notation beginnend mit dem Verzeichnispfad zu der Datei und dem eigentlichen Dateinamen. Der Dateiname selbst beginnt mit `lib`, dann folgt eine eindeutige Bezeichnung, die Dateiendung ist `.so`, als Abkürzung für *Gemeinsam genutztes Objekt; engl.: Shared Object*. Dieser *Linker Name* wird um eine Versionsnummer zum *soname* erweitert, die hochgezählt wird, wenn sich die Schnittstelle ändert. Für den Namen der tatsächlichen Bibliotheksdatei, die den eigentlichen Code enthält, wird der *soname* nochmals um eine so genannte *Minor Number* und eine optionale Veröffentlichungsnummer ergänzt. Üblicherweise werden die zusätzlichen Dateien aus Verweisen (*softlink*) aufgebaut, d.h. der *Linker Name* verweist auf den *soname*, der wiederum

auf den tatsächlichen Bibliotheks-Dateinamen verweist. Dieses System ermöglicht eine effiziente Kontrolle der auf einem System installierten Versionen gemeinsam genutzter Bibliotheken.

GCC Parameter

Um mit der GCC eine gemeinsam genutzte Bibliothek zu erstellen, müssen eine Reihe Aufrufparameter übergeben werden.

Positionsunabhängiger Code

Eine Voraussetzung für die Erstellung einer gemeinsam genutzten Bibliothek ist es, dass die Zielplattform positionsunabhängigen Code unterstützt. Der für eine Bibliothek zu verwendende Objektcode muss in diesem Fall in positionsunabhängiger Form vorliegen, da erst zum Zeitpunkt des Ladens einer Bibliothek feststeht, an welcher Stelle im Arbeitsspeicher welche Bestandteile abgelegt werden. Beispielsweise können auf diese Weise Funktionsaufrufe aus bereits geladenen Bibliotheken verwendet werden und globale Variablen aus allen geladenen Bibliotheken referenziert werden.

Zugriffe auf globale Referenzen werden indirekt über eine *Globale Offset Tabelle* (*engl.: Global Offset Table – GOT*) durch den dynamischen Linker aufgelöst.

Um Quellcode in positionsunabhängigen Objektcode zu übersetzen, verwendet die GCC die Parameter `-fpic` und `fPIC`, wobei `-fpic` eher kompakteren Code als `-fPIC` produziert und die plattformabhängige Maximalgrösse der *GOT* berücksichtigt, indem eine Fehlermeldung zurückgegeben wird, falls die *GOT* zu gross wird. Mit `-fPIC` wird die Maximalgrösse der *GOT* umgangen.

Mit dem folgenden Aufruf der GCC wird der Code der oben beschriebenen Schnittstelle in Objektcode übersetzt, der für die Einbindung in eine dynamische Bibliothek vorbereitet ist.

```
arm-linux-gcc -c -fpic durchschnitt.c
```

Erstellen der Bibliothek

Die entstandene Objektdatei `durchschnitt.o` kann nun (eventuell mit weiteren Objektdateien) zu einer gemeinsam genutzten Bibliothek zusammengesetzt werden. Dafür benötigen die GCC einige Parameter:

```
arm-linux-gcc -shared -Wl,-soname,libdurchschnitt.so.1 \
 -o libdurchschnitt.so.1.0.1
```

`-shared` zeigt an, dass eine gemeinsam genutzte Bibliothek erstellt werden soll.

`-Wl,-soname,libdurchschnitt.so.1` übergibt die durch Komma getrennten Parameter `-soname` und `libdurchschnitt.so.1` an den Linker. Dadurch wird dem Linker der *soname* mitgeteilt.

`-o` gibt mit `libdurchschnitt.so.1.0.1` den Dateinamen der zu erzeugenden Bibliothek an.
`durchschnitt.o` spezifiziert die Eingabedateien mit dem Objektcode.

Die so erstellte Bibliothek kann nun auf einer ARM-Plattform installiert, mittels `ldconfig` konfiguriert und verwendet werden.

ldconfig

Das Werkzeug `ldconfig` ist bei der Verwaltung installierter Bibliotheken behilflich. Es durchsucht die Verzeichnisse `/lib` und `/usr/lib`, sowie die in der Datei `/etc/ld.so.conf` und der Kommandozeile aufgeführten Verzeichnisse nach neuen Bibliotheks-Dateien und erstellt die entsprechenden *soname* Verweise. Die Verweise für die *Linker Name*-Dateien müssen allerdings manuell angelegt werden, was es ermöglicht, eine bestimmte Version der Bibliothek zu verwenden.

Die Liste der gefundenen Dateien wird standardmässig in der *Cache*-Datei `/etc/ld.so.cache` gespeichert, was ein effizientes Auffinden einer Bibliotheks-Datei zur Laufzeit ermöglicht.

Für den Entwicklungszeitraum ist es hilfreich, Bibliotheken nicht sofort im System verfügbar machen zu müssen, sondern die entsprechende Konfiguration in einem lokalen Verzeichnis vorzunehmen. Mit dem Parameter `-n` kann das Verzeichnis angegeben werden, in dem sich die übersetzten Bibliotheken befinden, so dass `ldconfig` die entsprechenden Verweise von den *soname*-Dateien auf die Bibliotheksdateien erstellen kann. Danach müssen noch die Verweise auf die *soname*-Dateien ohne Versionsnummer erstellt werden.

```
ldconfig -n .
ln -s libdurchschnitt.so.1 libdurchschnitt.so
```

Das ergibt folgende Dateien:

```
libdurchschnitt.so -> libdurchschnitt.so.1
libdurchschnitt.so.1 -> libdurchschnitt.so.1.0.1
libdurchschnitt.so.1.0.1
```

LD_LIBRARY_PATH

Besonders für Entwicklungszwecke ist die Verwendung der Umgebungsvariable `LD_LIBRARY_PATH` sinnvoll. Über sie können, jeweils getrennt durch Doppelpunkte, zusätzliche Bibliotheken angegeben werden, die vor den im System konfigurierten geladen werden sollen.

ldd

Mit dem Befehl `ldd` werden die Bibliotheken angezeigt, die ein Programm für seine Ausführung benötigt. Als Parameter wird der Programmname angegeben, dessen Bibliotheken ausgegeben werden sollen.

```
ldd verbrauch
linux-gate.so.1 =>  (0xb7f99000)
libdurchschnitt.so.1 => ./libdurchschnitt.so.1 (0xb7f95000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e34000)
/lib/ld-linux.so.2 (0xb7f9a000)
```

Dynamisches Laden

Beim dynamischen Laden einer gemeinsam genutzten Programmbibliothek wird der Programmcode der Bibliothek nicht beim Start des Programms in den Arbeitsspeicher geladen, sondern erst zu dem Zeitpunkt, an dem das Programm die Bibliothek benötigt. In C dient dazu die Funktion `dlopen()` mit folgender Signatur:

```
void * dlopen(const char *filename, int flag);
```

Der Parameter `filename` enthält den Dateinamen der zu ladenden Bibliothek und mit `flag` wird der Zeitpunkt spezifiziert, zu dem nicht definierte Funktionsnamen aufgelöst werden sollen. Als Werte können entweder `RTLD_LAZY` oder `RTLD_NOW` angegeben werden. Mit `RTLD_LAZY` wird erst dann nach den Funktionen gesucht, wenn der Code in der Bibliothek ausgeführt wird, was zu einem Laufzeitfehler führt, wenn die passende Funktion nicht gefunden wird. Bei `RTLD_NOW` werden alle Funktionsaufrufe aufgelöst, während `dlopen()` ausgeführt wird.

Das Gegenstück zu `dlopen()` ist `dlclose()`, womit signalisiert wird, dass die Bibliothek nicht mehr benötigt wird und aus dem Arbeitsspeicher entfernt werden kann, sobald kein anderes Programm mehr auf sie zugreift.

Diese Vorgehensweise hat den Vorteil, dass die Bibliothek nicht über die gesamte Laufzeit des Programms im Speicher gehalten werden muss und dass der Programmstart schneller erfolgen kann. Allerdings dauert der erste Zugriff auf die Bibliothek länger, da diese zuerst in den Speicher geladen werden muss.

6.2.1.4 Ausführbare Programme

Ausführbare Programme (engl.: Executable Programs) können vom Betriebssystem gestartet werden und werden dann in einem eigenständigen Prozess ausgeführt, in dem der Programm-Code unabhängig von und parallel zu den übrigen Prozessen durchlaufen wird. Dazu erhält jeder Prozess einen definierten Satz an Systemressourcen, auf die er zugreifen kann.

GCC Parameter

Für den Erstellungsvorgang benötigt der Präprozessor der GCC Zugriff auf die C Header-Dateien, um verwendete Schnittstellendeklarationen in den Quellcode einzufügen. Falls es sich um Header-Dateien handelt, die weder in einem Standardverzeichnis der GCC noch relativ zu den einbindenden Dateien liegen, kann der Suchpfad mit dem Parameter `-I<Verzeichnis>` entsprechend erweitert werden.

Ansonsten benötigt der Linker der GCC für den Übersetzungsvorgang die Namen der dynamischen Bibliotheken, die aus dem Programm heraus angesprochen werden, was mit dem Parameter `-l<Bibliotheksname>` geschieht. Falls sich die Bibliotheken nicht im Suchpfad des Systems befinden, werden dem Linker mittels `-L<Verzeichnis>` weitere Verzeichnisse angegeben, in denen nach dynamischen Bibliotheken gesucht werden soll. Das Beispiel zur Berechnung des Durchschnittsverbrauchs aus der Datei `verbrauch.c` wird mit folgender Anweisung in eine ausführbare Datei übersetzt.

```
arm-linux-gcc -L. -ldurchschnitt -o verbrauch verbrauch.c
```

6.2.2 Java

Die Programmiersprache und Laufzeitumgebung *Java* wurde von einem Team um James Gosling bei der inzwischen von Oracle übernommenen Firma SUN Microsystems Anfang der neunziger Jahre entwickelt und 1995 der Öffentlichkeit vorgestellt. Der ursprüngliche Einsatzzweck für Java war eine sogenannte Set-Top Box, die ein Fernsehgerät um zusätzliche Funktionalität wie z. B. Video-On-Demand erweitert. Ziel war es, ein System zu entwickeln, dass auf unterschiedlichster Hardware bei gleichzeitig beschränkter Rechenkapazität einsatzfähig ist. Insofern war Java von Anfang an ideal geeignet für den Einsatz im Pervasive Computing, da hier dieselben Anforderungen gestellt werden. Die Programmiersprache Java ist vollständig objektorientiert, lehnt sich mit der Syntax bei C++ an, hat aber einige Konzepte aus anderen Sprachen wie z. B. Smalltalk übernommen.

Am 13 November 2006 wurde der Java Standard von SUN Microsystems unter der General Public License, Version Zwei (GPLv2) veröffentlicht. Damit wurde diese hardwareunabhängige Programmiersprache und Plattform der Open-Source-Gemeinde zur Verfügung gestellt.

Die Spezifikation von Java sieht eine Mischung aus interpretierten und übersetzten (kompilierten) Konstrukten vor. Der Java Quellcode wird in eine hardwareunabhängige Maschinensprache, den sogenannten *Bytecode*, übersetzt. Dieser Bytecode wird dann von einer virtuellen Maschine (VM) auf der Ziel-Hardware interpretiert und ausgeführt. Abbildung 6.1 zeigt die für die Erstellung und Ausführung von Java-Programmen notwendigen Schritte.

Durch die Übersetzung in das hardwareunabhängige Zwischenformat des Bytecodes wird einerseits Plattformunabhängigkeit erreicht, andererseits aber auch ein Performanzgewinn im Vergleich zu vollständig interpretierten Sprachen erzielt, da der Bytecode bereits in Maschinensprache vorliegt und sich somit der Interpretationsaufwand reduziert.

Ein weiterer für den Einsatz im Pervasive Computing interessanter Aspekt ist der geringe Speicherverbrauch von Java Bytecode. Selbst umfangreichere Programme sind oft nur mehrere Kilobyte gross, was dem eingeschränkten Ressourcenangebot natürlich entgegenkommt.

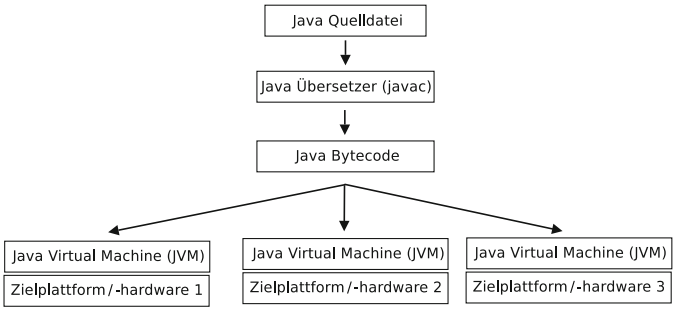

Abb. 6.1 Die Funktionsweise der Java Plattform

6.2.2.1 Umgebungsvarianten

Inzwischen wurden Weiterentwicklungen des Java Standards veröffentlicht und in drei unterschiedliche Einsatzbereiche unterteilt. Die drei Umgebungen unterscheiden sich hauptsächlich hinsichtlich des jeweiligen Funktionsumfangs und des entsprechenden Ressourcenbedarfs:

Java Standard Edition (Java SE): Dies ist die Basisdefinition von Java, die die zentralen Klassenbibliotheken mit dem Funktionsumfang für den Einsatz auf gängigen Arbeitsplatzrechnern festlegt.

Java Enterprise Edition (Java EE): Für den serverseitigen Einsatz wurde die Java SE Plattform um Elemente erweitert, mit denen sich robuste und skalierbare Anwendungen erstellen lassen, die auf Anwendungsservern zentral im Netz ausgeführt werden. Architekturstandards wie Java Servlets und Enterprise Java Beans (EJB) sind teil dieser Spezifikation.

Java Micro Edition (Java ME): Speziell für ressourcenbeschränkte Umgebungen des Pervasive Computing wurde die Micro Edition von Java definiert.

Eine entscheidende Voraussetzung für den Einsatz von Java ist eine virtuelle Maschine für die Zielhardware. Im Bereich der Java Standard Edition (Java SE) gibt es eine grosse Auswahl an verfügbaren virtuellen Maschinen. Oracle stellt die Referenzimplementierung der Java VM u.a. für Linux zum Download bereit. Mittlerweile wurde auch der Quellcode unter dem Namen *OpenJDK* auf http://openjdk.java.net/ veröffentlicht. Die Apache Software Foundation hat mit dem Projekt *Harmony* eine von Oracle unabhängige quelloffene Implementierung der Java Plattform zum Ziel (http://harmony.apache.org). Darüber hinaus gibt es auch einige kommerzielle Anbieter für Java VMs auf Linux, die grösstenteils ebenfalls kostenlos erhältlich sind.

Java Micro Edition (Java ME)

Die Architektur der Java Plattform hat grösstmögliche Plattformunabhängigkeit zum Ziel. Die Funktionalitäten von Geräten des Pervasive Computing decken ein

derart weites Spektrum unterschiedlicher Einsatzgebiete und Geräteeigenschaften ab, das es unmöglich macht, alle Anforderungen mit einer einzigen Spezifikation zu erfüllen. Aus diesem Grund wurde für Java ME ein modularer Ansatz gewählt.

Zielgeräte werden aufgrund ihrer Eigenschaften und Einsatzgebiete gruppiert und für jede Gruppe wird in einer Konfiguration (engl.: Configuration) festgelegt, in welchen Punkten die Java Plattform von der Java SE Spezifikation abweicht, d.h. welche für Java SE definierte Funktionalität in der konkreten Java ME Konfiguration nicht zur Verfügung steht. Diese Vorgehensweise ist ein praktikabler Kompromiss zwischen dem Anspruch, eine einzige Plattform für unterschiedlichste Hardware bereitzustellen und der Gerätevielfalt mit teilweise extrem eingeschränkter Ressourcenkapazität.

Konfigurationen beschreiben die grundlegenden Geräteeigenschaften, wie Prozessorleistung, verfügbarer Hauptspeicher und die Qualität einer vorhandenen Netzwerkverbindung. Für eine Konfiguration werden dann so genannte Profile definiert, die den Einsatzzweck eines Gerätes genauer beschreiben, z. B. ob es sich um ein Mobiltelefon oder einen PDA handelt. Momentan sind zwei Java ME Konfigurationen spezifiziert, die in den folgenden Abschnitten beschrieben werden.

Zusätzlich zur Klassifizierung in Form von Konfigurationen und Profilen kann eine Java ME Umgebung weitere optionale Pakete bereitstellen, um die Plattform um Funktionalität für spezielle Einsatzzwecke zu erweitern. Beispielsweise ist die Verwendung von Bluetooth-Netzwerken in einem derartigen Paket gekapselt.

Abbildung 6.2 veranschaulicht die Zusammenhänge der Java ME Architektur.

Connected Device Configuration (*CDC*)

Diese Java ME Konfiguration beschreibt verhältnismässig leistungsfähige Geräte, wie z. B. gut ausgestattete PDAs, Telematiksysteme in Fahrzeugen oder so genannte TV Set-Top-Boxen. Geräte dieser Leistungsklasse verfügen über einen 32 Bit-Prozessor, stellen der Java VM mindestens 2MB Speicher zur Verfügung und sind mit einer leistungsfähigen Netzwerkverbindung ausgestattet.

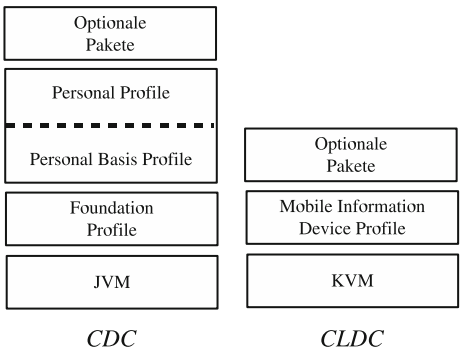

Abb. 6.2 Übersicht über die Java ME Plattform

Die CDC beinhaltet eine vollständige Java VM und grosse Teile der Java SE Sprachbibliotheken. Für CDC sind z. Z. drei Profile spezifiziert, die aufeinander aufbauen und miteinander kombiniert werden können:

Foundation Profile (FP): Das Grundprofil definiert hauptsächlich den Netzwerkzugriff und ist für Geräte ausreichend, die keine grafische Benutzeroberfläche benötigen.

Personal Basis Profile (PBP): Mit dem Basisprofil werden einfache grafische Elemente für die Darstellung einer Oberfläche bereitgestellt.

Personal Profile (PP): Das Personenprofil beinhaltet das Basisprofil (PBP) und die komplette Sprachbibliothek für grafische Elemente aus der Java SE Spezifikation, das so genannte Abstract Window Toolkit (AWT). Mit diesem Profil können grafische Anwendungen realisiert werden, wie sie aus dem Arbeitsplatzumfeld bekannt sind.

Connected Limited Device Configuration (*CLDC*)

Die Konfiguration CLDC wurde für extrem ressourcenbeschränkte Geräte spezifiziert, die nur über eine Netzwerkverbindung mit geringer Bandbreite und nicht ständiger Verbindung verfügen. Die Spezifikation fordert von Geräten dieser Leistungsklasse einen 16-Bit oder 32-Bit Prozessor und mindestens 192 KB Speicher für die Java VM.

Geräte dieser Kategorie sind Mobiltelefone und einfache PDAs. In der Tat dürften Mobiltelefone der Haupteinsatzzweck für diese Konfiguration sein.

Es existiert momentan nur ein Profil für CLDC, das so genannte Mobile Information Device Profile (MIDP). Dieses Profil spezifiziert den Netzwerkzugriff, einfache grafische Bedienelemente sowie die Möglichkeit, Benutzerdaten persistent abzuspeichern.

Die Java VM in CLDC, die so genannte KVM, verfügt nur über einen kleinen Teil der in J2SE definierten Funktionalität. Der Name KVM soll andeuten, dass diese VM nur wenige Kilobyte Speicher verbraucht, das K steht in diesem Fall für Kilobyte.

Java Standard Edition (Java SE) for Embedded

Seitdem auch eingebettete Systeme immer leistungsfähiger werden und der Aspekt der Ressourcenknappheit auf diesen Systemen immer mehr in den Hintergrund tritt, werden auch die auf diesen Geräten ausgeführten Anwendungen immer anspruchsvoller. Heutige PDAs und Mobiltelefone können nicht nur Adressen verwalten und Telefonverbindungen herstellen, sondern nebenbei Videos abspielen und fotografieren. Für dieses Segment der leistungsfähigen mobilen Geräte wurde eine Version der Java SE entwickelt, die sich von der ursprünglichen Java SE durch Verzicht auf ressourcenintensive Bestandteile wie die grafische Bibliothek „Swing“, als auch durch die Unterstützung zusätzlicher Plattformen unterscheidet. Als Richtwert für

den Einsatz von Java SE for Embedded werden Geräte genannt, die mindestens über 32MB RAM und 32MB nichtflüchtigen Speicher (ROM/Festplatte/Flash) verfügen.

In diese Kategorie fallen die meisten gängigen PDAs und Mobiltelefone.

6.2.2.2 Strukturierung

Zur Organisation von Programmcode hat sich die von der Java-Spezifikation empfohlene Namenskonvention bewährt. Jede Klasse in Java muss einem Paket zugeordnet werden, fehlt diese Information wird das Standardpaket verwendet. Daher beginnt der Quellcode einer Java-Datei in der Regel mit der Definition des Paketnamens.

```
package com.springer.durchschnitt;
```

Dabei wird der Internetdomänenname der Organisation, in der oder für die das Programm entwickelt wird, rückwärts aufgeführt und mit projektspezifischen Unterkategorien erweitert. Mit Hilfe dieser Konvention wird ein Namensraum (engl.: Name Space) für den Quellcode definiert, über den er eindeutig referenziert werden kann.

6.2.2.3 Schnittstellen

Auch Java bietet die Möglichkeit, die Deklaration von Funktionalität von deren Implementierung zu trennen. Dazu dient das `interface` Konstrukt. In einer Schnittstellendatei werden sämtliche öffentliche Operationen inklusive ihrer Ein- und Rückgabewerte deklariert, wobei zwischen Schnittstellen im Gegensatz zu Klassen Mehrfachvererbung möglich ist. Analog zu dem Beispiel in C, deklariert die folgende Datei `Durchschnitt.java` eine Java-Schnittstelle für die Berechnung des Durchschnitts aus zwei float-Parametern.

```
package com.springer.durchschnitt;

public interface Durchschnitt
{
  float durchschnitt(float liter, float kilometer);
}
```

Die Java-Klasse `DurchschnittStandard.java` implementiert die Schnittstelle beispielhaft.

```
package com.springer.durchschnitt;

public class DurchschnittStandard implements Durchschnitt
{
  public float durchschnitt(float liter, float kilometer)
  {
    return liter / kilometer;
  }
}
```

Der Übersetzungsvorgang wird mit dem Programm `javac` durchgeführt, das Teil jeder Java Entwicklungsumgebung ist.

```
javac com/springer/durchschnitt/DurchschnittStandard.java
```

Der Übersetzer löst dabei die Abhängigkeiten zwischen den Quelldateien auf, d.h. dass in diesem Beispiel nicht nur die Java-Klasse `DurchschnittStandard` übersetzt wird, sondern auch die Schnittstelle `Durchschnitt`, falls diese nicht schon in Bytecode-Form vorliegt.

Ein einfaches Beispiel für ein lauffähiges Programm, das die Implementierung der Durchschnittsfunktionalität nutzt, sieht folgendermassen aus.

```
package com.springer.durchschnitt;

public class Verbrauch
{
  public static void main(String args[])
  {
    Durchschnitt durchschnitt = new DurchschnittStandard();
    System.out.println("Verbrauch pro 100 Kilometer: "
      + durchschnitt.durchschnitt(Integer.parseInt(args[0]),
                                  Integer.parseInt(args[1]))
      * 100);
  }
}
```

6.2.3 Perl

Perl wurde von Larry Wall ursprünglich zur Verarbeitung von Texten entwickelt. Es handelt sich um eine Skriptsprache, deren Quellcode zur Laufzeit interpretiert und ausgeführt wird. Das Projekt präsentiert sich auf der Webseite http://www.perl.org/. Beim Entwurf von Perl standen Einfachheit und Praxisnähe im Vordergrund. Die Syntax ist an die Sprache C (Abschn. 6.2.1) angelehnt, versucht aber, sich möglichst an menschlicher Sprache zu orientieren. Entwickelt wurde Perl ursprünglich für UNIX-Systeme und findet sich daher auf praktisch jedem Linux-System.

Der hohe Verbreitungs- und Bekanntheitsgrad von Perl machen die Sprache interessant sowohl während der Entwicklung als auch als Zielumgebung auf einem Pervasive Linux System. Beispielsweise benötigen die GNU Autotools (Abschn. 5.2.6) Perls als Voraussetzung.

Perl-Skripte haben üblicherweise die Dateiendung `.pl`. Der Interpreter wird mit dem Befehl `perl` aufgerufen, als z. B. `perl verbrauch.pl` oder mit folgender Zeile im Skript deklariert:

```
#!/usr/bin/env perl
```

Funktionsbibliotheken werden bei Perl *Module* genannt.

6.2.4 Python

Python ist eine objektorientierte, plattformübergreifende Programmiersprache und wurde von Guido van Rossum entworfen. Python-Programme werden interpretiert, d.h. der Quellcode wird nicht für eine bestimmte Hardware-Plattform übersetzt, sondern zur Laufzeit vom Interpreter ausgeführt. Das setzt vorraus, dass der Interpreter für die Zielplattform übersetzt werden kann bzw. verfügbar ist. Die Entwicklung von Python wird von der Python Software Foundation auf der Webseite http://www.python.org/ koordiniert.

Einen weiteren Schwerpunkt legt Python auf die Integration mit anderen Programmiersprachen. Der Name leitet sich ab aus der BBC Fernsehreihe *Monty Python's Flying Circus*.

Als Interpreter lässt sich Python entweder interaktiv oder mit einer Befehlsdatei aufrufen, die einen Satz an Kommandos enthält, die der Reihe nach ausgeführt werden sollen. Im Fall der interaktiven Betriebsart werden die Kommandos von der Standardeingabe eingelesen.

Ausserdem kann der Python-Interpreter mit einem einzelnen Kommando als Parameter aufgerufen werden. Dazu dient der Parameter `-c`.

Als grafische Entwicklungsumgebung dient *IDLE*, die selbst komplett in Python geschrieben ist.

Aufgrund der hohen Integrationstiefe mit anderen Programmiersprachen und dadurch, dass Python-Programme interpretiert werden, bietet sich die Sprache für den Einsatz im Pervasive Computing an.

6.2.4.1 Syntax

Die Syntax von Python ist stark an die Programmiersprache C (Abschn. 6.2.1) angelehnt. Phython ist schwach typisiert, d.h. Variablen können im Programmablauf unterschiedliche Typen annehmen. Im Gegensatz zu Sprachen wie C oder Java werden in Python Funktionsblöcke nicht in Klammern eingefasst, sondern durch Einrückung kenntlich gemacht.

6.2.4.2 Interaktiver Modus

Im interaktiven Modus erwartet der Python-Interpreter nach dem Start mit `python3` (für die Version 3 des Interpreters) den nächsten auszuführenden Befehl, üblicherweise mit der Eingabeaufforderung `>>>`.

```
$ python3
Python 3.1.2 (r312:79147, Apr 15 2010, 12:35:07)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for \
 more information.
>>>
```

Benötigt eine Zeile zusätzliche Informationen zur Vervollständigung, z. B. in geschachtelten Blöcken, wird dies durch drei Punkte `...` signalisiert.

Beendet wird der interaktive Modus durch den Befehl `quit()`.

6.2.4.3 Skript Modus

Python-Programme können direkt als ausführbare Skript-Dateien abgelegt werden. Dazu muss das Programm (im Fall von Version 3) mit folgender Zeile beginnen:

```
#! /usr/bin/env python3.1
```

Die Dateiendung von Python-Programmen ist `.py`. Mit dem Parameter `-m` wird der Interpreter mit einem Python-Skript gestartet.

```
python3 -m verbrauch.py
```

6.2.5 JavaScript

Die Skript-Sprache *JavaScript* wird im Internet zunehmend eingesetzt, um flexible und dynamische Webseiten zu erstellen, wobei das Interpretieren, also die Verarbeitung der Anwendungslogik, auf das mobile Gerät verlagert wird. JavaScript wurde von der Firma Netscape entwickelt und im Jahr 1995 veröffentlicht. Der Sprachkern ist als *ECMAScript* inzwischen auch von der ISO standardisiert.

Bis auf die an die Sprache C angelehnte Syntax hat JavaScript nicht viel mit der Programmiersprache Java (Abschn. 6.2.2) gemeinsam. JavaScript ist eine objektbasierte interpretierte Skriptsprache, die in den meisten Fällen innerhalb eines Web-Browsers ausgeführt wird. Nach einer längeren Phase, in der JavaScript fast in Vergessenheit geraten war, erlebt die Sprache eine Renaissance in modernen Web-Anwendungen, gerade im Zusammenhang mit dem für das Pervasive Computing wichtigen Standard HTML5 (Abschn. 6.7.2). Mit Hilfe von JavaScript können Webseiten Programmcode auf dem Browser im Zielgerät ausführen, unabhängig davon, ob eine Netzwerkverbindung zu dem Server besteht, von dem die Seite geladen wurde.

6.3 Entwicklungsumgebung

In diesem Kapitel wird die Einrichtung einer Entwicklungsumgebung für die Programmiersprache C Schritt für Schritt an einem Beispiel beschrieben. Sobald eine Toolchain installiert ist und mit der Entwicklung begonnen werden soll, muss die Entwicklungsumgebung eingerichtet, also Suchpfade für einzubindende Quell- und Binärbibliotheken definiert werden. Ausserdem müssen benötigte Konfigurationsdateien erstellt werden. In diesem Kapitel geht es darum, die Arbeitsumgebung für die Software-Entwicklung zu konfigurieren. Ziel ist es, die in Abschn. 6.2.1

beschriebene dynamische geladene Programmbibliothek reproduzierbar zu konfigurieren und zu übersetzen.

Üblicherweise suchen die GCC in den Verzeichnissen `/usr/local/include` und `/usr/include` nach den Header-Dateien und entsprechend in den Verzeichnissen `/usr/local/bin` und `/usr/bin` nach den binären Bibliotheken.

Wie in Abschn. 6.2.1.3 ausgeführt, wird wiederverwendbare Funktionalität in Form von Programmbibliotheken zur Verfügung gestellt. Um diese nun in eigener Software verwenden zu können, müssen die Definitionen der Schnittstellen für den Übersetzer im Quellformat verfügbar sein und für den Linker in für die Zielplattform übersetzter Form.

Dabei bietet es sich an, sämtliche für eine Zielumgebung benötigten Bibliotheken in einem eigens dafür vorgesehenen Unterverzeichnis zu bündeln, so dass es auf dem Entwicklungsrechner nicht zu einer Mischung aus Bibliotheken für Entwicklungs- und Zielumgebung kommt. z. B. kann bei Quellcode, der mit einem von den GNU Autotools erstellten `configure`-Skript konfiguriert wird, das Installationsverzeichnis der Toolchain mit dem Parameter `--prefix` angegeben werden.

```
./configure --host=arm-angstrom-linux-gnueabi \
  --prefix=/usr/local/arm
```

Falls das Werkzeug pkg-config verwendet wird, muss es so konfiguriert werden, dass es nur Metainformationen von installierten Paketen ausliest, die für die Zielplattform übersetzt wurden. Standardmässig sucht `pkg-config` im Unterverzeichnis `lib/pkgconfig` des Verzeichnisses, in dem es installiert wurde. Mit der Umgebungsvariable `PKG_CONFIG_LIBDIR` kann dieser Standardsuchpfad auf das Verzeichnis eingestellt werden, in das die Bibliotheken der Zielplattform installiert wurden.

```
export PKG_CONFIG_LIBDIR=/usr/local/arm/lib/pkgconfig
```

Dem Übersetzer der GCC werden die Verzeichnisse mit folgenden Parametern bekanntgegeben:

-Idir Dieser Parameter spezifiziert ein Verzeichnis, welches an den Anfang der Liste der Verzeichnisse gesetzt werden soll, in denen die GCC nach Header-Dateien suchen.

-Ldir Mit diesem Parameter wird die Liste der Verzeichnisse, in denen nach Bibliotheken gesucht wird, erweitert. Das angegebene Verzeichnis wird vor den Verzeichnissen im Standardsuchpfad durchsucht.

Um die Konfiguration und die Übersetzungsschritte zu dokumentieren und reproduzieren zu können, bietet sich der Einsatz der GNU Autotools an.

6.3.1 Verzeichnisstruktur und Dateien

Sinnvollerweise werden die für ein Programm relevanten Dateien in typorientierten Unterverzeichnissen gruppiert. Prinzipiell ist es natürlich jedem Entwickler selbst überlassen, wie die Projektdateien organisiert werden. Der hier gemachte Vorschlag

orientiert sich an bewährten und etablierten Vorgehensweisen. Die folgenden Beispiele beziehen sich auf ein beliebiges Projekthauptverzeichnis.

Für die Quellcodedateien in diesem Beispiel wird das Unterverzeichnis `src` verwendet. In diesem Verzeichnis befinden sich die Dateien `durchschnitt.c` und `durchschnitt.h`.

Dateien, die nicht direkt Quellcode enthalten, also z. B. Konfigurations-, Bild- und sonstige Datendateien, werden im Verzeichnis `data` abgelegt.

Im Hauptverzeichnis des Projektes wird die Datei `configure.ac` platziert, die von dem Werkzeug Autoconf gelesen wird:

```
AC_INIT
AC_CHECK_HEADERS([gtk/gtk.h])
AC_CHECK_HEADERS([stdlib.h])
AC_CHECK_HEADERS([durchschnitt.h])
AC_OUTPUT
```

Mit `AC_INIT` wird zunächst die Autoconf-Infrastruktur initialisiert. Anschliessend wird mittels `AC_CONFIG_SRCDIR` sichergestellt, dass das korrekte Verzeichnis für Quelldateien spezifiziert wurde.

Jedes Unterverzeichnis, das Quellcodedateien oder andere Dateien enthält, die bearbeitet und installiert werden sollen, muss eine Datei `Makefile.in` enthalten. Dies sind die Eingabedateien für das `configure`-Skript, aus der die eigentlichen `Makefile`-Dateien erzeugt werden.

Zum Schluss muss mit `AC_OUTPUT` die Datei `config.status` generiert und ausgeführt werden. Dies ist eine Shell-Skriptdatei, die die eigentlichen Tests durchführt und zusätzliche Konfigurations- und Projektdateien erzeugt, die sich aus der Auswertung der Plattformtests ergeben.

Schliesslich kann mit dem Aufruf `autoconf` die Datei `configure` erzeugt und mit den übrigen Projektdateien ausgeliefert werden.

6.4 Interprozesskommunikation (IPC)

Damit Programme, die gleichzeitig (parallel) auf dem selben System in verschiedenen Prozessen ausgeführt werden, miteinander kommunizieren können, gibt es diverse Formen der *Interprozesskommunikation* (*engl.: Interprocess Communication – IPC*) auf unterschiedlichen Ebenen.

6.4.1 Linux Systemaufrufe

Linux selbst bietet aus dem UNIX-Bereich übernommene Konzepte auf Betriebssystemebene.

6.4.1.1 Signale

Über *Signale* können laufenden Prozessen bestimmte Ereignisse oder Zustandsänderungen angezeigt werden. Beispielsweise nutzt der Linux-Kernel Signale, um

einem Prozess Fehler bei der Programmausführung mitzuteilen. Signale haben sowohl einen eindeutigen Namen mit dem Präfix `SIG` als auch eine Identifikationsnummer mit einem positiven Zahlenwert. Der Befehl auf der Kommandozeile, um Signale an Prozesse zu senden, lautet `kill`. Die Liste der unterstützten Signale kann mit dem Parameter `-l` ausgegeben werden.

```
kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Prozesse können Signale entweder ignorieren (bis auf die Signale `SIGKILL` und `SIGSTOP`), mit spezieller Logik auf diese reagieren oder sie zur Standardbehandlung an den Kernel weiterleiten. Um auf Signale reagieren zu können, muss ein Prozess eine Signalbehandlungsfunktion (engl.: Signal Handler) implementieren und diese registrieren. Ursprünglich diente dazu die Systemfunktion `signal`. Diese erwartet als Parameter die Identifikationsnummer des Signals, auf das reagiert werden soll und die Adresse der Funktion, in der das Signal behandelt wird. Alternativ zu einer Behandlungsfunktion können die Werte `SIG_IGN` oder `SIG_DFL` übergeben werden, um das Signal zu ignorieren oder der Standardbehandlung zu übergeben. Die Funktion gibt beim Aufruf den Wert zurück, der bisher für das Signal registriert war. Allerdings wird empfohlen, statt `signal` die Funktion `sigaction` zu verwenden. Hintergrund dafür ist, dass `signal` für verschiedene Linux- und UNIX-Systeme und -versionen unterschiedlich implementiert wurde und sich dementsprechend unterschiedlich verhält. Das ist gerade für die Plattformenvielfalt bei Pervasive Linux ein entscheidender Nachteil. Ausserdem ist das Verhalten von `signal` in Prozessen, die mehrere Threads verwenden, nicht definiert. `sigaction` hat eine ähnliche Signatur wie `signal`. Zusätzlich zur Signalidentifikationsnummer und der Adresse der Behandlungsfunktion erwartet die `sigaction` einen Parameter, der den Wert der bisherigen Behandlungsfunktion erhält. Im Erfolgsfall gibt die Funktion den Wert der bisherigen Behandlungsfunktion zurück, `SIG_ERR` im Fehlerfall.

Zum Versenden von Signalen dient die Systemfunktion `kill` mit der Identifikationsnummer des Prozesses, an den das Signal geschickt werden soll, und der

Signalnummer als Parameter. Das Versenden von Signalen zwischen Prozessen ist beschränkt auf Prozesse, die von demselben Benutzer gestartet wurden, abgesehen natürlich von Benutzern mit *root*-Rechten, die Signale an beliebige Prozesse schicken können.

6.4.1.2 Stream Pipes

Mit (Um-)Leitungen (engl.: Pipes) kann die Standardausgabe eines Programms in die Standardeingabe eines anderen Programm geführt werden. Auf der Befehlszeile dient dazu das *Pipe*-Symbol |.

```
cat README | less
```

Damit wird der Inhalt der Datei `README` ausgelesen und an das Programm `less` übergeben, um ihn seitenweise anzuzeigen.

Halbduplex-Pipes

In C-Programmen werden Pipes mit der Funktion `pipe` erstellt. Die Funktion erwartet ein zweidimensionales Feld vom Typ `int`. Der Aufruf veranlasst den Kernel, zwei interne Dateideskriptoren zu erstellen und deren Nummern in das Feld zu schreiben. Diese Deskriptoren existieren nur innerhalb des Linux-Kernels, die Nummer eines Deskriptors für Leseoperationen wird in das erste Feldelement geschrieben, das zweite Element erhält die Nummer eines Schreibdeskriptors. Damit kann das Programm den zweiten Deskriptor zum Schreiben von Daten verwenden und denselben Inhalt über den ersten einlesen.

Für die Interprozesskommunikation macht sich diese Vorgehensweise den Umstand zunutze, dass Kindprozesse alle offenen Dateideskriptoren des Elternprozesses übernimmt und diese somit in beiden Prozessen zur Verfügung stehen. Wird nun also mit der Funktion `fork` ein Kindprozess zur parallelen Ausführung erstellt, können beide Prozesse über diese Dateideskriptoren kommunizieren, indem der eine Prozess auf zweiten Deskriptor schreibt und der andere aus dem ersten liest. Dabei ist zu beachten, dass die jeweils ungenutzen Deskriptoren von den Prozessen mittels `close` geschlossen werden sollen.

In diesem Fall kann der Datenfluss nur in eine Richtung erfolgen, daher wird hier von *Halbduplex*-Pipes gesprochen.

Das Funktionspaar `popen` und `pclose` vereinfacht die Verwendung von Pipes für bestimmte Fälle. `popen` benötigt den Namen eines Befehls, der auf der Kommandozeile in einem Kindprozess ausgeführt werden kann und die Information, ob dieser Kindprozess Daten vom Elternprozess liest oder an diesen schickt.

Named Pipes

Im Gegensatz zu den oben beschriebenen anonymen Halbduplex-Pipes, die nur zwischen Eltern- und Kindprozessen funktionieren, bieten benannte Pipes (engl.:

Named Pipes) grössere Flexibilität. Named Pipes werden über ihren Namen identifiziert und ihre Dateideskriptoren sind im Dateisystem sicht- und zugreifbar. Eine Named Pipe ist eine spezielle Datei, deren Schreib- und Leseoperationen dem *First In First Out* (*FIFO*) Prinzip folgt. Die Daten, die zuerst in die Datei geschrieben wurden, werden also mit der Leseoperation auch zuerst wieder ausgegeben.

Named Pipes werden mit der Funktion `mknod` im System erzeugt. Als Parameter erwartet die Funktion den Dateinamen, also den Ort im Dateisystem, an dem die Named Pipe erstellt werden soll, sowie den Modus der Datei. Der dritte Parameter enthält die Major- und Minor-Nummern für den Fall, dass eine Gerätedatei erstellt werden soll, ansonsten wird `0` übergeben.

```
mknod("/home/cc/SPRINGER_FIFO", S_IFIFO|0666, 0);
```

Mit diesem Befehl wird eine Named Pipe mit dem angegebenen Namen erstellt. Der Parameter `S_IFIFO` sorgt dafür dass der Modus `FIFO` verwendet wird, verbunden mit den Zugriffsberechtigungen `0666` für die Datei.

6.4.1.3 Shared Memory

Bei gemeinsam genutzten Speicherbereichen (engl.: *Shared Memory*) können mehrere Prozesse auf den selben Abschnitt im Arbeitsspeicher zugreifen, es handelt sich also um eine sehr direkte Art der Kommunikation. Der Datenaustausch über Shared Memory ist sehr effizient, birgt aber auch das Risiko von Datenverlust, wenn die Zugriffe auf den Speicherbereich nicht ausreichend koordiniert sind.

Unter Linux gibt es mehrere Implementierungen von Shared Memory.

BSD mmap

Die ursprünglich aus dem BSD System stammenden Funktionen erlauben es, Dateien aus dem Dateisystem auf Arbeitsspeicherbereiche abzubilden. Auf diese abgebildeten Dateien kann dann mit entsprechenden Berechtigungen aus unterschiedlichen Prozessen zugegriffen werden. Mit der Funktion `mmap` wird eine Datei auf einen definierten Speicherbereich abgebildet und mit `munmap` wieder freigegeben. Veränderungen am Inhalt dieses Speicherbereichs werden in der dazugehörenden Datei erst sichtbar, wenn die geänderten Inhalte mit `msync` explizit abgeglichen werden.

System V IPC

Die Interprozesskommunikation des System V verwendet die Konzepte von *Message Queues*, *Semaphoren* und *gemeinsam genutzten Segmenten*.

Gemeinsam Genutzte Segmente

Gemeinsam genutzte Speichersegmente (engl.: Shared Segments) bieten ähnlich wie BSD mmap die Möglichkeit, Arbeitsspeichersegmente zu adressieren und aus unterschiedlichen Prozessen darauf zuzugreifen. Mit den Funktionen `shmget`,

`shmat`, `shmctl` und `shmdt` wird auf Shared Memory zugegriffen. `shmget` erstellt ein neues Segment und `shmat` bildet es auf einen Speicherbereich ab. Die Zuordnung von Segment und Speicher wird mit `shmdt` wieder aufgelöst. Mit der Funktion `shmctl` kann das Segment modifiziert und auch gelöscht werden, wenn es von keinem Prozess mehr benötigt wird.

Semaphoren

Mit Semaphoren können mehrere Prozesse den Zugriff auf gemeinsam genutzte Ressourcen koordinieren. Mit Hilfe von Semaphoren kann ein Prozess die Verwendung einer Ressource signalisieren, und damit verhindern, dass andere Prozesse gleichzeitig auf diese Ressource zugreifen. Dabei gibt der Wert der Semaphore an, wie viele Ressourcen des angeforderten Typs im System noch verfügbar sind. Ein Prozess kann den Wert einer Semaphore prüfen und ihn um die gewünschte Anzahl verringern, vorausgesetzt, dass noch ausreichend viele Ressourcen verfügbar sind.

Die System-Funktionen für die Verwendung von Semaphoren sind `semget`, `semop` und `semctl`. Mit `semget` wird eine neue Semaphore erstellt, Operationen auf der erstellten Semaphore werden mit `semop` ausgeführt. Beispielsweise kann mit `semop` der Wert der Semaphore geändert werden, um die Benutzung der assoziierten Ressource anzuzeigen. Ebenso werden Semaphoren über `semop` auch wieder freigegeben, so dass andere Prozesse auf die Ressource zugreifen können. Üblicherweise wartet der Aufruf von `semop` so lange, bis die gewünschte Anzahl Ressourcen verfügbar ist. Die Ausführung der weiteren Programmlogik in diesem Prozess wird so lange angehalten, bis wieder ausreichend Ressourcen im System verfügbar sind. Mit `semctl` wird die Semaphore gelöscht oder anderweitig manipuliert.

Message Queues

Bei Nachrichtenwarteschlangen (engl.: *Message-Queues* können Daten von mindestens einem Prozess als Pakete in Form so genannter Messages in eine Warteschlange geschrieben und dort von einem oder mehreren Prozessen ausgelesen werden. An der Kommunikation können also mehrere Prozesse sowohl schreibend als auch lesend teilnehmen. Messages werden vom schreibenden Prozess mit einem Typ versehen, den lesende Prozesse zur Identifikation nutzen können.

Eine neue Warteschlange wird mit der Funktion `msgget` erstellt. Zum Versenden von Nachrichten dient die Funktion `msgsnd`, empfangen wird mit `msgrcv`.

6.4.1.4 Sockets

Sockets bieten die Möglichkeit, auf relativ niedriger Netzwerkebene über Rechnergrenzen hinweg Daten zwischen Prozessen auszutauschen.

Um eine Socket-Verbindung aufzubauen rufen beide Prozesse die Funktion `socket` auf, um jeweils einen Socket als Verbindungsendpunkt zu erstellen. Damit eine Verbindung zustandekommen kann, muss der erste Prozess den Socket mit der Funktion `bind` an eine Adresse binden und anschliessend auf eingehende Verbindungen warten. Dazu dient die Funktion `listen`. Sobald der zweite Prozess

eine Verbindung aufbaut, kann diese vom ersten Prozess mit `accept` akzeptiert werden. Die Funktion `accept` wartet, bis ein Verbindungsversuch stattfindet.

Der zweite Prozess muss die Adresse kennen, an die der erste seinen Socket gebunden hat. Mit dieser Information kann er über die Funktion `connect` die Verbindung zum ersten Prozess aufbauen.

Voraussetzung dafür, dass zwei Sockets verbunden werden können, ist, dass beide vom selben Typ sind und der gleichen Adressen-Domäne angehören. Mögliche Socket-Typen sind *Datenstrom Sockets* und *Datagramm Sockets*. Bei Datenstrom Sockets werden die Daten Zeichen für Zeichen übertragen, während im Fall von Datagramm Sockets eine Nachricht (engl.: Message) die Übertragungseinheit ist. Die Adressen-Domäne gibt an, wie die Adressen aufgebaut sind, an die Sockets gebunden werden. Die verbreitetste Adressen-Domäne ist die Internet-Domäne. Adressen bestehen hier aus der über das *Internet Protokoll* (*IP*) vergebenen eindeutigen Kennung (IP Adresse) und einer so genannten *Port-Nummer*. Sobald die Verbindung auf diese Weise zustandegekommen ist, können beide Prozesse über die Sockets lesen und schreiben.

6.4.2 D-Bus

D-Bus, zu finden auf http://freedesktop.org/wiki/Software/dbus ist ein Subsystem zur Vermittlung von Nachrichten zwischen parallel ausgeführten Anwendungen. D-Bus ermöglicht darüber hinaus den Start von Diensten, die für das Ausführen eines anderen Programms benötigt werden. Über D-Bus können Anwendungen über Veränderungen im System benachrichtigt werden. Zum Beispiel kann eine Anwendung auf das Hinzufügen von Geräten zum System reagieren.

D-Bus ermöglicht die Kommunikation zwischen Programmen, die auf dem selben System ausgeführt werden, sowie die eingeschränkte Kommunikation über Systemgrenzen hinweg. Als Subsystem zur Verteilung von Nachrichten wird D-Bus hauptsächlich zur Kommunikation zwischen Programmen einer Desktop-Umgebung verwendet.

Das Datenaustauschformat von D-Bus ist binär und für die Kommunikation auf einem System konzipiert. Für den Nachrichtenaustausch mit anderen Systemen wird die verwendete Byte Reihenfolge (Abschn. 6.6.2) in jeder Nachricht übermittelt. Diese effiziente Kommunikation macht den Einsatz von D-Bus im Pervasive Computing besonders interessant. Nachrichten werden zwischen Objekten und nicht zwischen Anwendungen ausgetauscht.

6.4.3 Web Services

Web Services stellen eine stark verbreitete Möglichkeit dar, Funktionsaufrufe bzw. Dienste über Rechnergrenzen hinweg, insbesondere über das Internet zu

ermöglichen. Die Architektur wird vom W3C unter http://www.w3.org/TR/ws-arch/ beschrieben. Dabei werden die vom Anbieter eines Web Service definierten Dienste in einer XML-Datei beschrieben, die den Konventionen der *Web Service Definition Language* (*WSDL*) folgt. Nutzer der angebotenen Dienste beschreiben den Aufruf ebenfalls als XML-Datei und schicken die so formulierte Anfrage an die Adresse des Web Service. Die konkrete Umsetzung des Web Service wertet die Daten in der Anfrage aus, führt die geforderte Funktion aus und schickt das Ergebnis ebenfalls als XML-Datei an den Aufrufer zurück. Zur Beschreibung der Nachrichten wird das *SOAP*-Protokoll verwendet, das ebenfalls vom W3C definiert wird: http://www.w3.org/TR/soap/. Auf diese Weise ergibt sich eine lose gekoppelte Kommunikation von Anwendungen, mit deren Hilfe sich eine *Service-Orientierte Architektur* (*SOA*) aufbauen lässt.

Die lose Kopplung hat den Vorteil, dass man sich nicht schon zum Entwicklungszeitpunkt an einen bestimmten Anbieter binden muss, sondern man zum Ausführungszeitpunkt bestimmen kann, welchen Dienst man in Anspruch nehmen will.

6.4.4 CORBA

Die *Common Object Request Broker Architecture* (*CORBA*) ist ein von der *Object Management Group* (*OMG*) definierter Standard zur systemunabhängigen Kommunikation zwischen Softwareobjekten, zu finden unter http://www.corba.org. Bei CORBA vermittelt ein so genannter Object Request Broker (ORB) zwischen den Objekten, die über diese Middleware kommunizieren.

Die Objekte werden hinter plattform- und sprachunabhängigen Schnittstellen verborgen, die mit einer speziellen Beschreibungssprache, der *Interface Definition Language* (*IDL*) spezifiziert werden. Die so beschriebenen Schnittstellen können mit einem speziellen Übersetzer, dem IDL-Compiler, auf Konstrukte der gewählten Implementierungssprache abgebildet werden. Das Übersetzungsergebnis sind Quellcodefragmente sowohl für die Implementierung der Objektfunktionalität, das so genannte *Skeleton*, als auch für den Zugriff von anderen Objekten aus über den *Stub*.

Objekte werden zur Laufzeit innerhalb eines ORBs mit einer eindeutigen Kennung identifiziert, der so genannten *Interoperable Object Reference* (*IOR*). Über die IOR kann ein Prozess das Objekt, das möglicherweise einem anderen Prozess zugeordnet ist, lokalisieren und diesem Nachrichten schicken bzw. von ihm empfangen. Wichtig ist, dass die beteiligten Objekte mit unterschiedlichen Programmiersprachen implementiert sein können, einzige Voraussetzung ist, dass die gewünschte Sprache vom eingesetzten ORB unterstützt wird.

Für die Kommunikation über ORB-Grenzen hinweg wurde das *General Inter-ORB Protocol* (*GIOP*) definiert, dessen häufigste und wichtigste Implementierung das *Internet Inter-ORB Protokoll* (*IIOP*) ist.

6.5 Grafische Benutzerschnittstellen

Grafische Benutzerschnittstellen sollen den Anwender eines Systems auf intuitive und ansprechende Weise bei der Bedienung unterstützen. Zur Auswahl stehen eine Reihe an grafischen Oberflächen samt den zugehörigen Programmbibliotheken.

Prinzipiell gibt es zwei unterschiedliche Ansätze, eine Benutzerschnittstelle zu implementieren, einen deklarativen und einen programmatischen. Beide Ansätze haben ihre Vor- und Nachteile und können teilweise auch miteinander kombiniert werden.

Bei grafischen Oberflächen wird immer auf eine durch den Benutzer ausgelöste Aktion reagiert. Die Anwendung wartet meistens in einer Schleife auf neue Aktivitäten des Benutzers.

6.5.1 GTK+

Mit *GTK+* (Abschn. 3.6.5) lassen sich plattformunabhängige Programme entwickeln, die eine grafische Benutzerschnittstelle anbieten.

Die Architektur von GTK+ sieht einen Signalmechanismus zur Ablaufsteuerung vor. Jede Benutzeraktion auf der Oberfläche löst ein Signal innerhalb von GTK+ aus. Programme können für diese Signale Funktionen registrieren, die aufgerufen werden, sobald das entsprechende Signal auftritt. Durch den Aufruf von `gtk_main` wechselt die Ablaufsteuerung eines Programms in den Benachrichtigungszustand, in dem es nur noch über die registrierten Funktionen auf Signale reagiert.

Das Grundgerüst der meisten Anwendungen, die GTK+ verwenden, sieht folgendermassen aus. Es öffnet ein Fenster und wartet auf das Signal zum Schliessen des Fensters.

```
#include <gtk/gtk.h>

static void beende(GtkWidget *widget,
                   gpointer   data)
{
    gtk_main_quit();
}

int main(int   argc,
         char *argv[])
{
    GtkWidget *window;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

    g_signal_connect(G_OBJECT(window), "destroy",
                     G_CALLBACK(beende), NULL);
```

```
    gtk_widget_show(window);

    gtk_main ();

    return 0;
}
```

Zunächst werden die von GTK+ bereitgestellten Funktionsdeklarationen und Typdefinitionen eingebunden.

```
#include <gtk/gtk.h>
```

Die Funktion `destroy` sorgt dafür, dass der Benachrichtigungszustand des Programms wieder verlassen wird, indem `gtk_main_quit()` aufgerufen wird. Funktionen, die für eine Signalbenachrichtigung registriert werden sollen, müssen die beiden Parameter vom `GtkWidget*` und `gpointer` deklarieren. Es handelt sich dabei um einen Zeiger auf das grafische Bedienelement, das das Signal auslöste sowie mögliche Daten, die von dem Element übergeben werden sollen.

In diesem Beispiel werden beide Werte ignoriert.

```
static void destroy(GtkWidget *widget,
                    gpointer   data)
{
    gtk_main_quit();
}
```

Die Funktion `main` für den Programmstart deklariert zunächst eine Variable für das später erstellte Programmfenster und initialisiert die GTK+ Umgebung über den Aufruf von `gtk_init`.

```
GtkWidget *window;

gtk_init(&argc, &argv);
```

Anschliessend wird das Hauptfenster des Programms erzeugt und die Funktion `beende` für das Signal `destroy` registriert.

```
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

g_signal_connect(G_OBJECT(window), "destroy",
                 G_CALLBACK(beende), NULL);
```

Ohne den Aufruf von `gtk_main_quit()` in der Signalbehandlungsfunktion `beende` würde der Benachrichtigungszustand, in dem das Programm auf den Aufruf von registrierten Signalbehandlungsfunktionen wartet, nie verlassen werden und das Programm würde nicht ordentlich beendet.

Danach kann das erstellte Fenster angezeigt und in den Benachrichtigungszustand übergegangen werden, in dem auf Signale gewartet wird.

```
gtk_widget_show(window);

gtk_main();
```

Für die Erstellung des Programms mit der GCC werden die von GTK+ bereitgestellten Metainformationen mittels `pkg-config` (Abschn. 5.2.4) ausgewertet und als Parameter für den Übersetzer und Linker übergeben.

```
gcc beispiel.c -o programm 'pkg-config --cflags --libs gtk+-2.0'
```

Um dasselbe Programm z. B. für die ARM-Plattform von Openmoko zu übersetzen, muss der entsprechende Suchpfad für `pkg_config` gesetzt und die zugehörige Toolchain verwendet werden.

```
export PKG_CONFIG_PATH=/usr/local/openmoko/arm/arm-angstrom-linux
-gnueabi/usr/lib/pkgconfig
/usr/local/openmoko/arm/arm-angstrom-linux-gnueabi/bin/gcc beispi
el.c -o programm \
 'pkg-config --cflags --libs gtk+-2.0'
```

Bisher hat das Programm noch keine nutzbringende Programmlogik. Deshalb wird es im nächsten Schritt um grafische Ein- und Ausgabeelemente erweitert.

```
#include <gtk/gtk.h>
#include <stdlib.h>
#include "durchschnitt.h"

static void beende(GtkWidget *widget, gpointer data)
{
    gtk_main_quit();
}

void berechne(gpointer widget, gpointer data)
{
    GtkWidget **durchschnittElement = (GtkWidget**)data;
    char ergebnis[30];

    float verbrauch = durchschnitt(
       atoi(gtk_entry_get_text(
                (GtkEntry*)durchschnittElement[0])),
       atoi(gtk_entry_get_text(
                (GtkEntry*)durchschnittElement[1]))) * 100;

    sprintf(ergebnis, "%f", verbrauch);

    gtk_label_set_text(
       (GtkLabel*)durchschnittElement[2], ergebnis);
}
```

```
int main(int   argc,
         char *argv[])
{
    GtkWidget *fenster;
    GtkWidget *v_box;
    GtkWidget *liter_text;
    GtkWidget *kilometer_text;
    GtkWidget *berechnen;
    GtkWidget *durchschnitt_text;
    GtkWidget *durchschnitt[3];

    gtk_init(&argc, &argv);

    fenster = gtk_window_new(GTK_WINDOW_TOPLEVEL);

    g_signal_connect(G_OBJECT(fenster), "destroy",
                     G_CALLBACK(beende), NULL);

    v_box = gtk_vbox_new(TRUE, 2);

    gtk_container_add(GTK_CONTAINER (fenster), v_box);
    gtk_widget_show(v_box);

    liter_text = gtk_label_new("Liter:");
    gtk_container_add(GTK_CONTAINER (v_box), liter_text);
    gtk_widget_show (liter_text);

    durchschnitt[0] = gtk_entry_new();
    gtk_container_add (GTK_CONTAINER (v_box),
                       durchschnitt[0]);
    gtk_widget_show (durchschnitt[0]);

    kilometer_text = gtk_label_new("Kilometer:");
    gtk_container_add(GTK_CONTAINER (v_box),
                      kilometer_text);
    gtk_widget_show(kilometer_text);

    durchschnitt[1] = gtk_entry_new();
    gtk_container_add(GTK_CONTAINER (v_box),
                      durchschnitt[1]);
    gtk_widget_show (durchschnitt[1]);

    berechnen = gtk_button_new_with_label("Berechne");
    gtk_container_add(GTK_CONTAINER (v_box), berechnen);
    gtk_widget_show (berechnen);

    durchschnitt_text = gtk_label_new("Durchschnitt:");
    gtk_container_add(GTK_CONTAINER (v_box),
                      durchschnitt_text);
    gtk_widget_show(durchschnitt_text);

    durchschnitt[2] = gtk_label_new("0");
    gtk_container_add(GTK_CONTAINER (v_box),
```

```
                    durchschnitt[2]);
    gtk_widget_show(durchschnitt[2]);

    gtk_widget_show(fenster);

    g_signal_connect(berechnen, "clicked",
      G_CALLBACK(berechne), durchschnitt);

    gtk_main ();

    return 0;
}
```

Dabei nutzt es die in Abschn. 6.1 beschriebene einfache Bibliothek zur Berechnung einer Division.

Zunächst werden die benötigten Header-Dateien importiert.

In der Funktion `main` werden zuerst alle Variablen deklariert und dann die GTK-Umgebung initialisiert. Die Funktion `gtk_window_new` erstellt das Hauptfenster und anschliessend wird die Funktion `beende` für die Beendigung der GTK-Signalbehandlungsschleife registriert.

Eine `GtkVBox` ist ein unsichtbares GTK-Element für die Darstellung und Anordnung grafischer Bedienelemente. Alle einer GtkVBox hinzugefügten Bedienelemente werden vertikal untereinander angezeigt. Mit dem Aufruf der Funktion `gtk_vbox_new(TRUE, 2)` wird ein entspechendes Element erzeugt. Die an die Funktion übergebenen Parameter `TRUE` und `2` sagen aus, dass jedes grafische Bedienelement gleich gross dargestellt werden soll und dass dazwischen ein Abstand von 2 Pixeln besteht.

Alle folgenden Elemente werden mit der Funktion `gtk_container_add` dem jeweiligen übergeordneten Element hinzugefügt und mit `gtk_widget_show` sichtbar gemacht.

Jedes benötigte grafische Bedienelement wird über die zugehörige Funktion erzeugt. Die Eingabefelder werden in einem Feld (engl.: Array) zusammengefügt.

Die Funktion `berechne` hat die Aufgabe, die vom Benutzer des Programms eingegebenen Werte auszulesen und das berechnete Ergebnis im entsprechenden Ausgabefeld anzuzeigen. Dazu wird die Funktion für das Signal `clicked` des grafischen Druckknopfes registriert. Als Übergabeparameter erhält die Funktion einen Zeiger auf das grafische Bedienelement, das das Signal auslöste und einen Zeiger auf einen frei wählbaren Speicherbereich. Im Fall der Funktion `berechne` wird das Feld übergeben, das Zeiger auf die Bedienelemente enthält, die notwendig sind, um die Benutzereingaben auszulesen, die Berechnung durchzuführen und das Ergebnis anzuzeigen.

Die folgende Darstellung zeigt dasselbe Programm in der Oberfläche GNOME und auf einem Openmoko Mobiltelefon.

Abb. 6.3 GTK unter GNOME und Openmoko

6.5.2 Enlightenment

Der Quellcode für das Projekt Enlightenment kann von der Webseite http://www.enlightenment.org/ heruntergeladen werden. Benötigt werden die Bibliotheken *Eina*, *Eet*, *Evas*, *Ecore*, *Efreet*, *Embryo*, *Edje* und *E_DBus*. Da die Bibliotheken teilweise aufeinander aufbauen, müssen sie in dieser Reihenfolge jeweils gebaut und installiert werden.

```
./autogen.sh
make
make install
ldconfig
```

Sofern die Bibliotheken in das standardmässig vorgesehene Verzeichnis installiert und systemweit verfügbar gemacht werden sollen, werden für die beiden letzten Befehle die entsprechenden Rechte benötigt.

Nach den *Enlightenment Foundation Libraries* (*EFL*) fehlen dann nur noch der eigentliche Window Manager *Enlightenment* und die Bibliothek mit Bedienelementen *Elementary*, die auf ähnliche Weise erstellt und installiert werden.

```
./configure
make
make install
ldconfig
```

Das Grundgerüst von Anwendungen für Enlightenment, die die Bibliothek Elementary nutzen, sieht folgendermassen aus:

```
#include <Elementary.h>

EAPI int elm_main(int argc, char **argv)
{
   elm_run();
   elm_shutdown();
   return 0;
}
ELM_MAIN()
```

Über die Header-Datei `Elementary.h` werden alle notwendigen Elementary-Deklarationen und abhängigen Header-Dateien eingebunden. Elementary schreibt vor, dass die Einstiegsfunktion `elm_main` mit obiger Signatur heisst und dass diese Funktion als erste Funktion im Programm aufgerufen wird. Das Makro `ELM_MAIN` sorgt für die übliche `main`-Funktion, die Initialisierung von Elementary und den Aufruf von `elm_main`. Mit `elm_run` wird die Ereignisschleife (engl.: Event Loop) gestartet. Ab diesem Zeitpunkt wartet das Programm auf Aktionen des Benutzers. Sobald die Ereignisschleife beendet wird, was in diesem Programmgerüst nicht vorgesehen ist, sorgt der Aufruf von `elm_shutdown` für eine geordnete Beendigung des Programms und der Freigabe beanspruchter Ressourcen.

Eine Anwendung zur Berechnung des durchschnittlichen Kraftstoffverbrauchs benötigt ein paar Zeilen mehr.

```
#include <Elementary.h>
#include <stdlib.h>
#include "durchschnitt.h"

static void beende(void *data, Evas_Object *obj,
 void *event_info)
{
   elm_exit();
}

static void berechne(void *data, Evas_Object *obj,
 void *event_info)
{
   Evas_Object **durchschnittElement = (Evas_Object**)data;
   char ergebnis[30];

   float verbrauch = durchschnitt(
      atoi(elm_entry_entry_get(durchschnittElement[0])),
      atoi(elm_entry_entry_get(durchschnittElement[1]))) * 100;

   sprintf(ergebnis, "%f", verbrauch);

   elm_label_label_set(durchschnittElement[2], ergebnis);
}

EAPI int elm_main(int argc, char **argv)
```

```
{
   Evas_Object *fenster;
   Evas_Object *hintergrund;
   Evas_Object *box;
   Evas_Object *liter_text;
   Evas_Object *kilometer_text;
   Evas_Object *berechnen;
   Evas_Object *durchschnitt_text;
   Evas_Object *durchschnitt[3];

   fenster = elm_win_add(NULL, "verbrauch", ELM_WIN_BASIC);
   elm_win_title_set(fenster, "Verbrauch");
   evas_object_smart_callback_add(fenster, "delete,request",
 beende, NULL);

   hintergrund = elm_bg_add(fenster);
   evas_object_size_hint_weight_set(hintergrund, 1.0, 1.0);
   elm_win_resize_object_add(fenster, hintergrund);
   evas_object_show(hintergrund);

   box = elm_box_add(fenster);
   evas_object_size_hint_weight_set(box, 1.0, 1.0);
   elm_win_resize_object_add(fenster, box);
   evas_object_show(box);

   liter_text = elm_label_add(fenster);
   elm_label_label_set(liter_text, "Liter:");
   evas_object_size_hint_weight_set(liter_text, 1.0, 1.0);
   elm_box_pack_end(box, liter_text);
   evas_object_show(liter_text);

   durchschnitt[0] = elm_entry_add(fenster);
   evas_object_size_hint_weight_set(durchschnitt[0], 1.0, 1.0);
   evas_object_size_hint_align_set(durchschnitt[0], -1.0, -1.0);
   elm_entry_single_line_set(durchschnitt[0], EINA_TRUE);
   elm_box_pack_end(box, durchschnitt[0]);
   evas_object_show(durchschnitt[0]);

   kilometer_text = elm_label_add(fenster);
   elm_label_label_set(kilometer_text, "Kilometer:");
   evas_object_size_hint_weight_set(kilometer_text, 1.0, 1.0);
   elm_box_pack_end(box, kilometer_text);
   evas_object_show(kilometer_text);

   durchschnitt[1] = elm_entry_add(fenster);
   evas_object_size_hint_weight_set(durchschnitt[1], 1.0, 1.0);
   evas_object_size_hint_align_set(durchschnitt[1], -1.0, -1.0);
   elm_entry_single_line_set(durchschnitt[1], EINA_TRUE);
   elm_box_pack_end(box, durchschnitt[1]);
   evas_object_show(durchschnitt[1]);

   berechnen = elm_button_add(fenster);
   elm_button_label_set(berechnen, "Berechne");
```

```
    evas_object_size_hint_weight_set(berechnen, 1.0, 1.0);
    evas_object_size_hint_align_set(berechnen, -1.0, -1.0);
    elm_box_pack_end(box, berechnen);
    evas_object_show(berechnen);
    evas_object_smart_callback_add(berechnen, "clicked", berechne,
 durchschnitt);

    durchschnitt_text = elm_label_add(fenster);
    elm_label_label_set(durchschnitt_text, "Durchschnitt:");
    evas_object_size_hint_weight_set(durchschnitt_text, 1.0, 1.0);
    elm_box_pack_end(box, durchschnitt_text);
    evas_object_show(durchschnitt_text);

    durchschnitt[2] = elm_label_add(fenster);
    elm_label_label_set(durchschnitt[2], "0");
    evas_object_size_hint_weight_set(durchschnitt[2], 1.0, 1.0);
    evas_object_size_hint_align_set(durchschnitt[2], -1.0, -1.0);
    elm_box_pack_end(box, durchschnitt[2]);
    evas_object_show(durchschnitt[2]);

    evas_object_show(fenster);

    elm_run();
    elm_shutdown();
    return 0;
}

ELM_MAIN()
```

Dem Schema folgend wird wieder die Bibliothek zur Berechnung des Durchschnitts aus Abschn. 6.1 verwendet und die passende Header-Datei importiert. Die Header-Datei `stdlib.h` enthält die Funktion `atoi` zur Datentypumwandlung von `String` in `int`.

Die Funktionen `beende` und `berechne` enthalten die Logik zum Beenden der Ereignisschleife sowie zur Berechnung und Ausgabe des durchschnittlichen Kraftstoffverbrauchs. Diese Funktionen werden später als so genannte *Callback*-Funktionen an den entsprechenden Bedienelementen der Oberfläche registriert.

In der obligatorischen Funktion `elm_main` werden zuerst die Variablen für sämtliche Bedienelemente deklariert. Das Hauptfenster wird daraufhin mit dem Aufruf von `elm_win_add` erzeugt. Der erste Parameter gibt das Elternfenster an, was im Fall des Hauptfensters, also des ersten erzeugten Fensters einer Anwendung, nicht gegeben ist. Es folgen der Fenstername zur Identifikation und der Fenstertyp. Mit `elm_win_title_set` erhält das Fenster einen passenden Titel und `evas_object_smart_callback_add` registriert die erste Callback-Funktion, nämlich zur Beendigung des Programms, wenn der Benutzer in irgendeiner Form signalisiert, dass die Anwendung geschlossen werden soll, z. B. über eine entsprechende Schaltfläche in der Titelzeile oder eine definierte Tastenkombination.

Danach wird mit `elm_bg_add` ein Standard-Hintergrund für das Fenster erzeugt und mit `evas_object_size_hint_weight_set` festgelegt, dass sich die Grösse des Hintergrundobjektes verändern kann. Der Aufruf der Funktion `elm_win_resize_object_add` fügt das Hintergrundobjekt dem Fenster hinzu und `evas_object_show` zeigt es an.

Dieses Schema wiederholt sich nun für die restlichen Objekte.

Der Objekttyp *Box* sorgt für die Anordnung der Bedienelemente in einer Reihe, entweder vertikal oder horizontal, wobei vertikal der Standard ist. Zur Festlegung, dass ein Bedienelement am Schluss der Reihe hinzugefügt werden soll, dient der Aufruf von `elm_box_pack_end` Bei den Beschreibungsobjekten (text) wird noch der anzuzeigende Text mit `elm_label_label_set` gesetzt. Den Eingabefeldern und der Schaltfläche wird mit `evas_object_size_hint_align_set` signalisiert, dass sie die gesamte für sie verfügbare Breite und Höhe ausnutzen sollen und die Eingabe in den Feldern mit `elm_entry_single_line_set` auf eine Zeile beschränkt. Auf der Schaltfläche zur Berechnung des Verbrauchs muss dann noch die oben definierte Funktion mit `evas_object_smart_callback_add` registriert werden. Hierbei werden neben dem Ereignis, auf das reagiert werden soll (`clicked`) und der Funktion, die aufgerufen werden soll, noch die für das Auslesen der Eingabewerte und die Ausgabe des Ergebnisses benötigten Bedienelemente übergeben.

Das Ergebnis hat im einfachsten Fall das in Abb. 6.4 dargestellte optische Erscheinungsbild. Enlightenment sieht eine klare Trennung zwischen den Anwendungskomponenten und deren Erscheinungsbild vor. Die Optik, also Farb- und Formgestaltung der grafischen Elemente wird deklarativ über die Layout-Komponente Edje gesteuert. Die Beschreibung erfolgt mittels einer eigenen Layout-Sprache, die an den Standard *Cascading Style Sheets* (*CSS*) erinnert, der zur Gestaltung von Webseiten gang und gäbe ist.

Abb. 6.4 Enlightenment

6.5.3 GPE Palmtop Environment

GPE (Abschn. 3.6.11) basiert auf *GTK*, daher ist die Entwicklung von Software für GPE sehr ähnlich der für GTK.

6.5.4 Qt for Embedded Linux

Um Anwendungen für Qt for Embedded Linux zu entwickeln, muss zunächst der Programmcode der Bibliothek von der Webadresse http://qt.nokia.com/downloads heruntergeladen und mit Hilfe der Installationsanweisung für die Zielumgebung konfiguriert, übersetzt und installiert werden. Dem Quellcode liegen eine Reihe unterstützter Konfigurationen für diverse Plattformen und Betriebssysteme bei. Jede Konfiguration befindet sich in einem eigenen Unterverzeichnis unterhalb des Verzeichnisses `mkspecs`, beispielsweise befindet sich die Konfiguration für Linux auf einer ARM Plattform im Verzeichnis `mkspecs/qws/linux-arm-g++`.

Die Entwickler von Qt for Embedded Linux setzen für die Steuerung des Übersetzungsprozesses und die Entwicklung von Anwendungsprogrammen auf qmake (Abschn. 5.2.7). Das gilt auch für das Übersetzen der Qt-Bibliothek selbst, daher ist der Hauptbestandteil jeder Konfiguration die Datei `qmake.conf`. In dieser Datei werden die Toolchain- und Plattformabhängigkeiten gekapselt. Falls keine der mitgelieferten Konfigurationen für die zu verwendende Toolchain passt, wird empfohlen, eines der vorhandenen Verzeichnisse zu kopieren und für die eigene Umgebung anzupassen. Dabei ist es wichtig, dass der Verzeichnisname für die Konfiguration einer Linux-Plattform wie im obigen Beispiel mit `linux-` beginnt, da sonst der Übersetzungsvorgang fehlschlägt.

Folgendes Programmbeispiel in der Sprache C++ erzeugt ein Anwendungsfenster in einer Qt for Embedded Linux Umgebung:

```
#include <QApplication>
#include <QMainWindow>
#include <QWidget>

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  QWidget composite;
  QMainWindow mainWindow;

  mainWindow.setCentralWidget(&composite);
  mainWindow.show();
  return app.exec();
}
```

Die ersten drei Zeilen binden die Definitionen der im folgenden Ablauf benötigten Klassen ein.

```
#include <QApplication>
#include <QMainWindow>
#include <QWidget>
```

Jede Qt-Anwendung muss über genau ein Objekt vom Typ `QApplication` für die Verwaltung der Anwendungsressourcen verfügen. Aus der Klassendefinition `QMainWindow` wird das Anwendungsfenster erzeugt und `QWidget` wird als Platzhalterobjekt für das zentrale grafische Bedienelement benötigt, da die Klasse `QMainWindow` ein solches verlangt, selbst wenn es wie in diesem Beispiel keine Funktionalität bietet.

Mit den folgenden Zeilen wird zunächst das Platzhalterobjekt `composite` als zentrales Steuerelement für das Anwendungsfenster gesetzt, anschliessend wird selbiges auf der Oberfläche angezeigt und dann die Ausführungskontrolle an das Objekt vom Typ `QApplication` übergeben, welches auf Ereignisse auf der Benutzeroberfläche wartet.

```
mainWindow.setCentralWidget(&composite);
mainWindow.show();
return app.exec();
```

Alternativ hätte das Objekt `composite` auch ohne das Objekt `mainWindow` erzeugt werden können, da Objekte vom Typ `QWidget`, denen kein Elternobjekt zugeordnet wurde, automatisch in einem eigenen Fenster dargestellt werden.

6.5.5 Android

Wie in Abschn. 3.7.4 beschrieben, besteht Android aus einer mehrschichtigen Architektur und den dazugehörigen Entwicklungswerkzeugen. Die Installation unter Linux besteht aus dem Entpacken der SDK-Starter-Paketdatei in ein geeignetes Verzeichnis wie z. B. `/opt`.

```
cd /opt
tar -xvf /home/cc/Downloads/android-sdk_r08-linux_86.tar
```

Das SDK bringt ein Verwaltungswerkzeug mit, mit dem zusätzliche Komponenten für die Entwicklung heruntergeladen und installiert werden können. Das Programm *Android SDK and AVD Manager* befindet sich im Verzeichnis `tools` innerhalb des SDK-Verzeichnisses und wird mit dem Befehl `android` gestartet. Das Programm `android` kann auch über die Kommandozeile gesteuert werden, indem beim Aufruf entsprechende Parameter angegeben werden.

Um zusätzliche Komponenten zu installieren, wie z. B. unterschiedliche Android Zielumgebungen, bietet der Android SDK and AVD Manager zwei Quellen an: Das *Android Repository* und ein Verwaltungssystem für Drittanbieter (engl.: Third Party Add-Ons). Über das Auswahlmenü können die gewünschten Komponenten installiert werden.

Die Liste der installierten Android Zielumgebungen kann auf der Kommandozeile abgefragt werden:

```
android list targets
Available Android targets:
id: 1 or "android-9"
     Name: Android 2.3
     Type: Platform
     API level: 9
     Revision: 1
     Skins: QVGA, WVGA854, WVGA800, WQVGA432, HVGA (default), WQV
GA400
```

Der auf der Android-Webseite verfügbare Emulator erlaubt die Simulation unterschiedlicher Android-Plattformen (AVDs) samt deren optischer Erscheinungsbilder (Haut, engl.: Skin). Er basiert auf QEMU (Abschn. 5.3.1) und kann im Zusammenhang mit den ADT auch innerhalb von Eclipse genutzt werden.

Android-Anwendungen werden in der Programmiersprache Java (Abschn. 6.2.2) geschrieben und mit dem Übersetzer des Entwicklungssystems, wie im Java-Umfeld üblich, in plattformunabhängigen Bytecode übersetzt. Dieser Bytecode muss allerdings noch in so genannten `DEX` Bytecode überführt werden, der von der Dalvik-VM interpretiert werden kann. Diesen Schritt erledigt in der Regel das Werkzeug `dx` automatisch bei der Erstellung der Anwendung.

6.5.5.1 Android Virtual Devices

Virtuelle Abbilder von Android-Umgebungen werden entweder mit dem Android SDK and AVD Manager oder über die Kommandozeile des `android`-Werkzeugs erzeugt. Mit dem grafischen Werkzeug lassen sich sehr schnell AVDs erstellen

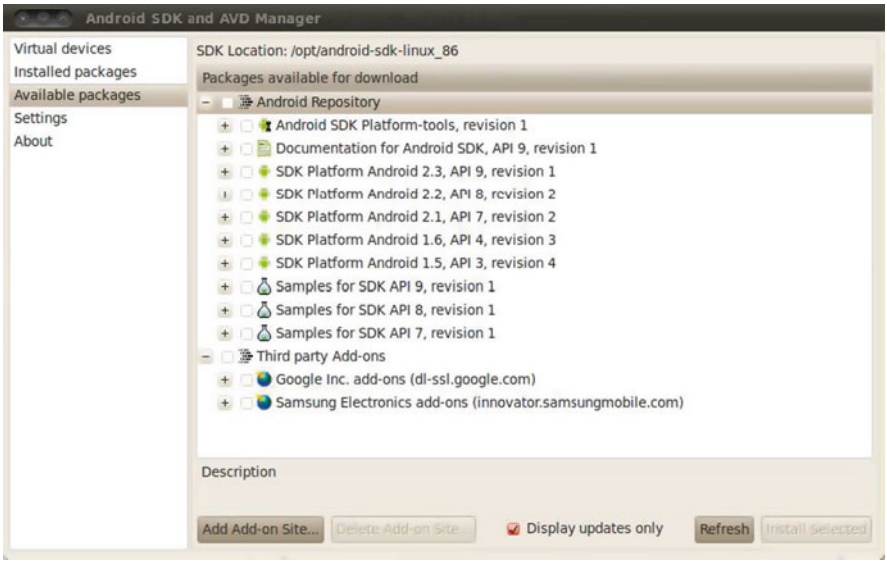

Abb. 6.5 Der Android SDK and AVD Manager

und konfigurieren. Standardmässig werden die Abbilder in einem Unterverzeichnis `/.android/avd/` im Heimatverzeichnis des aktuellen Benutzers abgelegt. Wenn ein anderes Verzeichnis verwendet werden soll, wird empfohlen, das AVD über die Kommandozeile zu erstellen.

```
./android create avd -n AndroidImage -t 1 -p ~/linux/android/
```

Der Parameter `-t` gibt die Nummer der zu verwendenden installierten Zielplattform an und der Parameter `-p` das Verzeichnis, in welches die Dateien des AVD gespeichert werden sollen. Dabei darf das Verzeichnis noch nicht existieren, das übergeordnete Verzeichnis muss dagegen vorhanden sein. Die Liste verfügbarer Zielplattformen kann über folgenden Befehl ermittelt werden.

```
android list target
Available Android targets:
id: 1 or "android-9"
     Name: Android 2.3
     Type: Platform
     API level: 9
     Revision: 1
     Skins: QVGA, WVGA854, WVGA800, WQVGA432, HVGA (default), WQV
GA400
```

Das Verschieben und Umbenennen eines AVDs oder der enthaltenden Verzeichnisstruktur ist insofern problematisch, als dass das AVD das Plattform-Abbild nur referenziert und nicht selbst enthält. Deswegen bietet das `android`-Werkzeug mit dem Befehl `move avd` einen entsprechenden Parameter an, um ein Abbild zu verschieben. Falls sich die übergeordnete Verzeichnisstruktur geändert hat, kann ein AVD mit dem Befehl `update avd` wieder lauffähig gemacht werden.

Sobald die Konfiguration beendet ist, kann die Emulation gestartet werden. Der Emulator simuliert dann ein Android-Gerät mit den zuvor konfigurierten Eigenschaften. Das optische Erscheinungsbild kann mit so genannten *Verkleidungen* (*engl.: Skin*) beeinflusst werden.

Das emulierte Gerät nutzt den Bildschirm und die Eingabegeräte des Systems, auf dem der Emulator ausgeführt wird. Ein Mausklick simuliert somit z. B. den Druck mit dem Finger auf einen berührungsempfindlichen Bildschirm und die Tastatur des emulierten Gerätes wird auf die des Rechners abgebildet, auf dem der Emulatur läuft.

Zur Kommunikation mit dem emulierten Gerät und zu dessen Steuerung dient die so genannte *Android Debug Bridge* (*adb*). Die Android Debug Bridge besteht aus einem Hintergrundprozess, der auf dem Entwicklungsrechner läuft, einer Komponente im Emulator, die die Steuerbefehle entgegennimmt und verarbeitet und einer Schnittstelle zum Benutzer bzw. zu Anwendungen wie den ADT. Für die Kommunikation zwischen einem ADB-Programm und dem Hintergrundprozess wird der TCP Port `5037` verwendet.

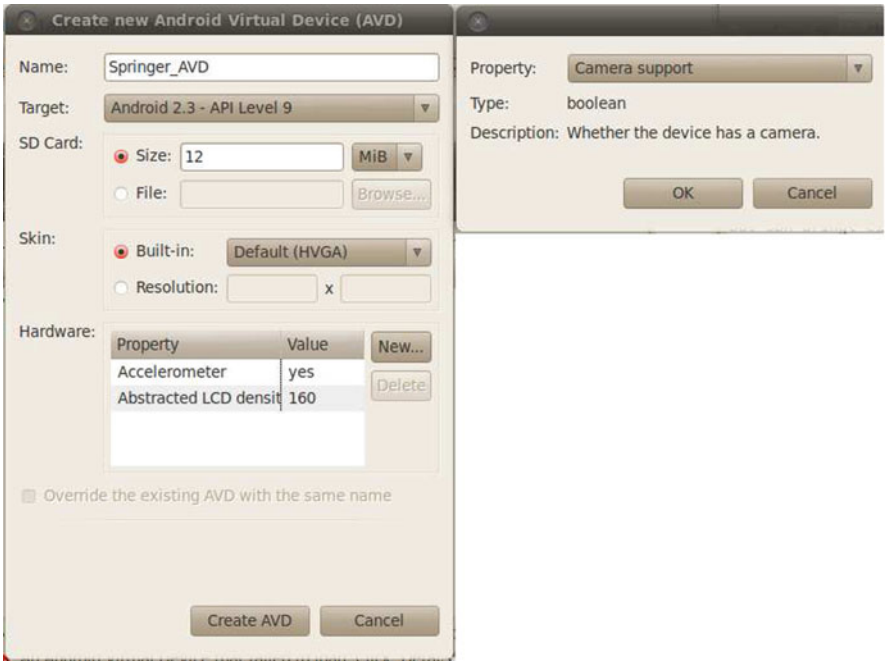

Abb. 6.6 Konfiguration eines AVD

Abb. 6.7 Der Andorid Hardware-Emulator

6.5.5.2 Android-Anwendungen

Anwendungen für die Android-Plattform werden im *Android Package*-Format mit der Dateiendung `.apk` ausgeliefert. Jede Android-Anwendung wird standardmässig in einer dedizierten Java-VM ausgeführt, die wiederum in einem eigenen Linux-Prozess läuft.

Die Android-Architektur sieht vier verschiedene Komponententypen einer Anwendung vor. *Aktivitäten* (*engl.: Activities*) stellen die sichtbare Schnittstelle zum Benutzer dar. Sie sind dafür verantwortlich, was dem Benutzer angezeigt wird. Aufgaben, die im Hintergrund ohne sichtbare Elemente ausgeführt werden, heissen *Dienste* (*engl.: Services*). Über *Sendungsempfänger* (*engl.: Broadcast Receivers*) kann eine Anwendung über Ereignisse und Veränderungen im System informiert werden, um darauf entsprechend zu reagieren. Mit *Datenversorgern* (*engl.: Content Providers*) kann eine Anwendung anderen Anwendungen Daten zur Verfügung stellen.

Bis auf Broadcast Receivers müssen alle Komponenten einer Anwendung in der Datei `AndroidManifest.xml` deklariert werden. Diese XML-Datei enthält Meta-Informationen über die Anwendung.

Abgesehen von den bereits vorgestellten Werkzeugen android, Android Emulator und Android Debug Bridge benötigt man für die Entwicklung von Programmen für die Android-Plattform noch die Hilfsmittel Ant (Abschn. 5.2.8), keytool und jarsigner. Ant wird für den automatisierten Erstellungsvorgang benötigt und die beiden Programme keytool und jarsigner sind Hilfsprogramme der Java SE Entwicklungsumgebung (Abschn. 6.2.2). Mit Hilfe von keytool werden private Schlüssel erstellt, mit denen die fertigen `.apk`-Dateien mittels jarsigner signiert werden.

Ein Entwicklungsprojekt für die Android-Plattform sieht folgende definierte Datei- und Verzeichnisstruktur vor: Die Verzeichnisse `src` für Quellcode, `bin` für die Ergebnisse des Erstellungsprozesses, `libs` für zusätzlich benötigte Bibliotheken, `res` für Ressourcen wie z. B. Bilddateien und `gen` für während des Erstellungsprozesses von Ant generierte Dateien. Im Hauptverzeichnis des Projekts befinden sich die Dateien `AndroidManifest.xml`, die die Metainformationen des Projektes enthält, die Steuerdatei `build.xml` für Ant, sowie die beiden Konfigurationsdateien `default.properties` und `build.properties` mit teilweise konfigurierbaren Eigenschaften für das Projekt.

Dieser Aufbau wird im Unterverzeichnis `tests` für Testzwecke dupliziert.

Das Anlegen der Struktur übernimmt `android` mit den Aufrufparametern `android create project`. Dabei werden folgende Parameter übergeben:

`--target`:	Als Wert wird die Nummer der installierten Android Zielumgebung angegeben, für die das Projekt erstellt werden soll.
`--path`:	Hier wird der Dateipfad des Verzeichnisses übergeben, in dem das Projekt angelegt wird.
`--activity`:	Dieser Wert bezeichnet den Namen der Java-Klasse des Projekts, die die Default-Activity implementiert, also typischerweise für den Programmstart aufgerufen wird.

`--package`: Mit diesem Parameter wird der Paketname anhand der im Java-Umfeld etablierten Konvention (Abschn. 6.2.2) festgelegt.

`--name`: Dieser Wert wird, falls er angegeben wird, als Name der `.apk`-Datei des Projektes verwendet. Ansonsten wird der Wert auch als Dateiname übernommen, der für `--activity` spezifizierte wurde.

Ein neues Projekt kann also beispielsweise folgendermassen angelegt werden:

```
android create project \
 --target 1 \
 --path ./verbrauch \
 --activity Verbrauch \
 --package com.springer.verbrauch.android
```

Dieser Aufruf erzeugt die beschriebene Verzeichnisstruktur sowie Initialversionen der essenziellen Dateien.

Wie bereits erwähnt werden Android-Anwendungen mit dem Werkzeug Ant gebaut. Eine Entwicklungsversion der Anwendung kann mit folgendem Aufruf erstellt werden.

```
ant debug
```

Für die zu veröffentlichende Produktionsversion dient folgender Befehl.

```
ant release
```

Wenn der Erstellungsvorgang erfolgreich ist, wird die fertige Anwendung in einem `.apk` Archiv abgelegt, deren Dateiname zuvor beim Anlegen des Projektes angegeben wurde.

Wichtig ist, dass jede Android-Anwendung mit einem Schlüssel vor der Installation signiert werden muss. Dafür werden die bereits genannten Werkzeuge keytool und jarsigner aus der Java Entwicklungsumgebung benötigt. Für Entwicklungsversionen (Debug) einer Anwendung ist allerdings auch ein entsprechender Entwicklungsschlüssel ausreichend. Bei der Erstellung einer Entwicklungsversion wird das resultierende Android-Archiv automatisch mit dem Entwicklungsschlüssel signiert. Produktionsversionen müssen allerdings manuell mit einem privaten Schlüssel signiert werden, bevor sie veröffentlicht werden können.

Um die Anwendung in einem AVD zu installieren, muss dieses AVD gestartet sein. Im Verzeichnis `platform-tools` des Android-SDKs findet sich das Werkzeug `adb`, die so genannte *Android Debug Bridge*, mit dem die gerade erstellte Android-Anwendung in ein AVD installiert werden kann, beispielsweise mit folgendem Befehl:

```
cd /opt/android-sdk-linux_86/platform-tools
./adb install /tmp/verbrauch/bin/Verbrauch-debug.apk

216 KB/s (13235 bytes in 0.059s)
pkg: /data/local/tmp/Verbrauch-debug.apk
Success
```

Damit wird die Android-Anwendung in den gestarteten Emulator installiert. Wenn mehr als ein Emulator auf demselben System gleichzeitig ausgeführt wird, muss zusätzlich die Seriennummer des Emulators angegeben werden, welche mit dem Aufruf `adb devices` ermittelt werden kann.

Für die Installation auf einem physikalischen Android-Gerät sind ein paar zusätzliche Schritte notwendig. Zum einen muss die Anwendung als Entwicklungsversion gekennzeichnet werden. Dazu wird in der das Paket beschreibenden Manifest-Datei `AndroidManifest.xml` in dem XML-Element `<application>` das Attribut `android:debuggable="true"` hinzugefügt wie im folgenden Beispiel:

```
<application android:label="@string/app_name"
 android:icon="@drawable/icon"
 android:debuggable="true">
```

In den ADT für Eclipse kann dasselbe über ein grafisches Auswahlfeld erfolgen. Ausserdem muss das Gerät, auf dem die Anwendung installiert werden soll, für das so genannte *USB Debugging* freigeschaltet werden. Dafür sieht die Android-Plattform eine spezielle Einstellung vor: Im Menü *Einstellungen* (*engl.: Settings*) kann über den Menübaum *Anwendungen* (*engl.: Applications*), *Entwicklung* (*engl.: Development*) der Menüpunkt *USB Debugging* aktiviert werden. Danach kann eine Anwendung über ein USB-Kabel installiert werden.

Abhängig von der verwendeten Linux-Distribution des Entwicklungssystems müssen noch Regeln definiert werden, anhand derer das System ein passendes Android-Gerät erkennt. Dazu wird eine neue Datei `61-android.rules` im Verzeichnis `/etc/udev/rules.d` angelegt mit einem von der Linux-Distribution und Version abhängigen Inhalt, wie z. B.

```
SUBSYSTEM=="usb", SYSFS{idVendor}=="0955", MODE="0666"
```

Die Herstelleridentifikation (engl.: Vendor ID), in diesem Fall `0955` für die Firma NVIDIA kann über den Befehl `adb devices` ermittelt werden.

Nun kann wie beim Emulator das Anwendungspaket installiert werden, wobei der Parameter `-d` angibt, dass auf einem über USB angeschlossenen Gerät installiert werden soll.

```
./adb -d install /tmp/verbrauch/bin/Verbrauch-debug.apk

248 KB/s (13235 bytes in 0.051s)
pkg: /data/local/tmp/Verbrauch-debug.apk
Success
```

Die Benutzeroberfläche einer Android-Anwendung kann sowohl deklarativ, so wie z. B. auch Webseiten mit HTML aufgebaut werden, als auch programmatisch erstellt werden, wobei sich die Namen der XML-Elemente und der zugehörigen Java-Klassen entsprechen. Ein `CheckBox` XML-Element wird dementsprechend auf die Klasse `CheckBox` abgebildet. Grafische Bedienelemente

werden im Android-Umfeld *Sichten* (*engl.: Views*) genannt. Diejenigen, mit denen der Anwender eine Aktion auslösen kann, wie z. B. Schaltflächen und Eingabefelder, heissen *Widgets*, für deren Anordnung wiederum Klassen vom Type `ViewGroup` zur Verfügung stehen.

Einer der Vorteile, die Oberfläche deklarativ aufzubauen, ist die explizite Trennung zwischen Benutzerschnittstelle und Anwendungslogik.

Der Aufbau der Benutzerschnittstelle geht dabei von genau einem Wurzelelement (engl.: Root Element) aus, unter dem sämtliche Bedienelemente einer Anwendung angeordnet sind. Alle weiteren Gestaltungselemente der Benutzeroberfläche werden logisch in eine Baumhierarchie eingegliedert.

Bei der initialen Erstellung eines Android-Projektes wird eine Datei `main.xml` im Verzeichnis `res/layout` angelegt, die ein minimales Gerüst für eine Benutzerschnittstelle definiert. Der Einstiegspunkt für den Aufbau der Benutzeroberfläche ist die Methode `onCreate` in der Standard-Activity-Klasse des Anwendungspakets. Hier wird über die Methode `setContentView` der Startpunkt für die Benutzeroberfläche gelegt. Um nun die eigene Layout-Datei `verbrauch_layout.xml` statt der generierten `main.xml` zu verwenden, muss der Programmcode an dieser Stelle entsprechend abgeändert werden. Der Aufruf

```
setContentView(R.layout.main);
```

wird durch

```
setContentView(R.layout.verbrauch_layout);
```

ersetzt.

Für die Layout-Dateien ist zu beachten, dass laut Namenskonvention nur Ziffern und Kleinbuchstaben für Dateinamen zu verwenden sind. Layout-Dateien werden üblicherweise im Verzeichnis `res/layout` des Android-Projektes gespeichert. Die nun nicht mehr benötigte Layout-Datei `main.xml` kann gelöscht werden.

Für das Beispiel der einfachen Anwendung, die den durchschnittlichen Kraftstoffverbrauch eines Fahrzeuges berechnet, sieht die Deklaration der Benutzerschnittstelle beispielsweise wie folgt aus:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text_liter"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Liter:" />
```

```
    <EditText android:id="@+id/edit_liter"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"/>
    <TextView android:id="@+id/text_kilometer"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Kilometer:" />
    <EditText android:id="@+id/edit_kilometer"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"/>
    <Button android:id="@+id/druckknopf_berechne"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Berechne" />
    <TextView android:id="@+id/text_verbrauch"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Durchschnitt:" />
    <TextView android:id="@+id/text_verbrauch_wert"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content" />
</LinearLayout>
```

Auf die bisher noch funktionslose Oberfläche kann nun mit Anwendungslogik zugegriffen werden. Zunächst wird in der Klasse `Verbrauch` eine innere Klasse vom Typ `OnClickListener` definiert. Diese hat die Aufgabe, darauf zu reagieren, wenn der Benutzer die Schaltfläche *Berechne* betätigt. Dazu werden in dieser Klasse zunächst alle benötigten Elemente der Benutzerschnittstelle über die Methode `findViewById` ausfindig gemacht. Als Parameter dienen die zuvor in der Datei `verbrauch_layout.xml` deklarierten Kennungen. Die Ein- und Ausgabefelder werden in ihre entsprechenden Typen überführt (engl: Class Cast), die Eingabewerte ausgelesen und die Berechnung durchgeführt. Das Ergebnis wird in dem Textfeld `verbrauchText` ausgegeben.

Die bei der Projekterstellung generierte Methode `onCreate` wird abgesehen von der ausgetauschten Layout-Datei nur um die beiden Zeilen erweitert, mit denen die gerade beschriebene innere Klasse auf der Schaltfläche registriert wird.

```
final Button berechne =
 (Button) findViewById(R.id.druckknopf_berechne);
berechne.setOnClickListener(berechneListener);
```

Der vollständige Quellcode ist etwas komplexer:

```
package com.springer.android.verbrauch;

import android.app.Activity;
import android.os.Bundle;
```

```
import android.view.View.OnClickListener;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import com.springer.durchschnitt.DurchschnittStandard;

public class Verbrauch extends Activity
{
    private OnClickListener berechneListener =
      new OnClickListener()
    {
        public void onClick(final View view)
        {
            final EditText literText =
              (EditText) findViewById(R.id.edit_liter);
            final EditText kilometerText =
              (EditText) findViewById(R.id.edit_kilometer);
            final TextView verbrauchText =
              (TextView) findViewById(R.id.text_verbrauch_wert);
            final DurchschnittStandard durchschnitt =
              new DurchschnittStandard();

            verbrauchText.setText(Float.toString(
              durchschnitt.durchschnitt(
                Float.parseFloat(literText.getText().toString()),
                Float.parseFloat(
                  kilometerText.getText().toString()))
                * 100));
        }
    };

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.verbrauch_layout);

        final Button berechne =
          (Button) findViewById(R.id.druckknopf_berechne);
        berechne.setOnClickListener(berechneListener);
    }
}
```

Die Klasse `R` wird beim Erstellungsvorgang für das Projekt vom Werkzeug `aapt` aus den Ressourcen-Dateien generiert und enthält u.a. die Namen der deklarierten Elemente der Benutzeroberfläche.

Abbildung 6.8 zeigt das Ergebnis im Android-Emulator.

Abb. 6.8 Die Kraftstoffverbrauchanwendung unter Android

6.6 Portabilität (von C-Programmen)

Die Programmiersprache *C* an sich ist plattformunabhängig, jedoch bedarf es einiger Umsicht bei der Programmierung, um in C geschriebene Programme portierbar zu halten. Die Spezifikation von C lässt Entwicklern von Übersetzern einigen Spielraum, der bei der Programmierung berücksichtigt werden muss.

Die folgenden Abschnitte erläutern einige wichtige Aspekte, die bei der Programmierung in C zu Portabilitätsproblemen führen. Eine umfassende Behandlung des Themas findet sich u.a. auf der Seite http://www-ccs.ucsd.edu/c/index.html.

6.6.1 Vorzeichenbehaftung des Datentyps char

Die Spezifikation von C sagt über die Vorzeichenbehaftung des Datentyps `char` nichts aus. Abhängig von Übersetzer, Betriebssystem oder Hardware- Architektur können Werte vom Type `char` entweder vorzeichenbehaftet (engl.: signed), also mit negativem und positivem Wertebereich, oder nicht vorzeichenbehaftet (engl.: unsigned), mit ausschliesslich positivem Wertebereich interpretiert werden. Aus diesem Grund sollte über die tatsächliche Vorzeichenbehaftung keine Annahme gemacht werden. Wird der Datentyp `char` als nicht vorzeichenbehaftet behandelt, ergibt ein Vergleich mit einem negativen Wert immer `0`. Folgendes Programm zeigt das Problem:

```
int main(void)
{
  char c = -1;
  printf("Der Wert von c ist: %d\n", c);

  if (c == -1)
  {
    printf("Der Datentyp char ist vorzeichenbehaftet!\n");
  }
  return 0;
}
```

Wird das Programm für ein System übersetzt, das den Datentyp char als vorzeichenbehaftet interpretiert, ist die Ausgabe wie folgt:

```
Der Wert von c ist: -1
Der Datentyp char ist vorzeichenbehaftet!
```

Auf einem System, das den Datentyp char als nicht vorzeichenbehaftet behandelt, ergibt das Programm folgendes Ergebnis. Der Vergleich mit `-1` ergibt in diesem Fall `0` und der Anweisungsblock der `if`-Verzweigung wird nie ausgeführt:

```
Der Wert von c ist: 255
```

Um hier für erhöhte Portabilität zu sorgen, sollte bei der Deklaration von Variablen des Datentyps `char` immer auch die Vorzeichenbehaftung angegeben werden, also `signed char` oder `unsigned char`. Für den Fall `signed char` gilt das nur dann, wenn der durch das Vorzeichen um ein Bit reduzierte Wertebereich für das konkrete Problem ausreichend ist. Andernfalls muss auf den Datentyp `int` ausgewichen werden.

6.6.2 Byte Reihenfolge (Endianness)

Abhängig von der Hardware-Architektur des Zielsystems werden Zahlen, die ausserhalb des Wertebereichs des Datentyps `byte` (8 Bit) liegen, unterschiedlich im Speicher repräsentiert. Die Zahl wird zerlegt in Einheiten der Grösse eines Byte und diese Einheiten werden hintereinander in den reservierten Speicherbereich geschrieben. Abhängig davon, ob mit dem höchstwertigen Byte (engl.: Most Significant Byte) begonnen wird oder mit dem niedrigstwertigen Byte (engl.: Least Significant Byte), wird die Vorgehensweise *Big Endian* bzw. *Little Endian* genannt.

Zu Portabilitätsproblemen führt dieser Unterschied einerseits, wenn Binärdaten zwischen Systemen ausgetauscht werden, die unterschiedliche Byte-Reihenfolge verwenden, und andererseits, wenn über Zeigerarithmetik auf Teilbereiche eines Wertes im Speicher zugegriffen wird, z. B. durch Type-Casting zwischen Zeigern auf Datentypen mit unterschiedlichem Wertebereich.

Um dem ersten Problem zu begegnen, muss in der Spezifikation des Dateiformates bzw. Netzwerkprotokolls festgelegt werden, in welcher Byte-Reihenfolge Binärdaten abgelegt werden oder die Datei enthält an einer definierten Stelle diese Information. Entsprechend muss dann auf Systemen mit der jeweils anderen Byte-Reihenfolge vor dem Schreiben bzw. nach dem Lesen eine Umwandlung der Byte-Reihenfolge vorgenommen werden.

Das zweite Problem tritt auf Big Endian Systemen auf. Soll z. B. die Adresse des Wertes eines Datentyps, der vier Byte Speicherplatz beansprucht einem Zeiger eines Datentyps, der zwei Byte Speicherplatz benötigt, zugewiesen werden, muss der Zeiger des vier Byte Datentyps um zwei Byte erhöht werden, da sich an der Adresse die beiden höherwertigen Byte des vier Byte Datentyps befinden. Das gleiche gilt natürlich auch umgekehrt, wenn der Wert eines zwei Byte Datentyps an die Adresse eines vier Byte Datentyps geschrieben werden soll. Derartiger Code muss Rücksicht auf die Hardware-Architektur nehmen und ist schwer portierbar.

6.6.3 Grösse des Datentyps Integer

Die Sprachdefinition von C schreibt nicht vor, aus wieviel Bit ein Wert des Datentyps *Integer* besteht. Es ist dem jeweiligen Übersetzer überlassen, wie der Datentyp für die Zielplattform effizient optimiert wird.

Die Internationale Standardisierungsorganisation (ISO) definierte im Jahr 1999 einen Sprachumfang von C als ISO C99, in dem u.a. Integer Datentypen mit garantierter Grösse definiert wurden: `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int32_t`, `uint32_t`. Die Verwendung dieser vordefinierten Datentypen macht insbesondere Programme für das Pervasive Computing leichter portierbar.

6.6.4 Calling Convention

Unter *Calling Convention* versteht man die Art und Weise, wie Parameter an Funktionen übergeben, weiterverarbeitet und wo die Ergebnisse abgelegt werden. Welche Calling Convention verwendet wird, hängt von der eingesetzten Software und auch der Prozessorarchitektur ab.

Die Calling Convention definiert die Schnittstelle zwischen Anwendungsentwicklung und Plattform.

6.6.5 uClinux

Für die Anwendungsentwicklung für Systeme, die über keine Speicherverwaltungseinheit verfügen und mit *uClinux* betrieben werden, gelten besondere Bedingungen. Unter uClinux gibt es keine dynamische Speicherverwaltung, so dass

Speicheradressen, die von Programmen verwendet werden, nicht unter Beibehaltung der für das Anwendungsprogramm bekannten Speicheradressen im physikalischen Speicher verschoben werden können.

6.6.5.1 Grösse des Stack

Die Grösse des Speicherbereiches für den Stack wird unter Linux normalerweise abhängig vom bereits beanspruchten Speicher dynamisch vergrössert. Unter uClinux jedoch muss die Grösse des benötigten Speichers für den Stack zum Zeitpunkt des Übersetzens festgelegt werden.

6.6.5.2 fork()

Die Funktion `fork()` ermöglicht das zeitgleiche Ausführen von Programmlogik. Durch den Aufruf von `fork()` wird der gerade ausgeführte Prozess dupliziert. Vater- und Kindprozess laufen unabhängig voneinander weiter. uClinux unterstützt die Funktion `fork()` nicht und bietet als Alternative die Funktion `vfork()` an. Mit dieser Funktion wird die Ausführung des Vaterprozesses unterbrochen und der Kindprozess fortgeführt, bis der Kindprozess `exec()` aufruft, um ein weiteres Programm zu starten, oder mittels `_exit` die eigene Ausführung beendet.

6.6.5.3 Shared Libraries

Die Erstellung und Verwendung von Shared Libraries erfordert in einem System ohne Speicherverwaltungseinheit besondere Aufmerksamkeit. In diesem Fall fehlt die Abstraktionsschicht zwischen tatsächlicher Speicheradresse und logischer Referenz im ausführbaren Code. Aus diesem Grund müssen beim Laden des Programms die Referenzen aufgelöst und die tatsächlichen Speicheradressen lokalisiert werden.

Shared Libraries werden unter uClinux mit einer eindeutigen Identifikationsnummer versehen, die sich auch in der Namenskonvention für die Dateinamen wiederspiegelt. Die Namen beginnen wie gewohnt mit `lib`, gefolgt von der numerischen Identifikation, beispielsweise `lib1.so`.

Dem Übersetzer wird die Verwendung von Identifikationsnummern mit der Option `-mid-shared-library` bekanntgegeben.

6.7 Web Anwendungen

Viele mobile Geräte können auf das Internet zugreifen und bieten eine Browser-Implementierung für die Darstellung von Webseiten. Das ermöglicht den Einsatz von *Web-Anwendungen* auf diesen Geräten, was wiederum die Portabilitätsproblematik vom Betriebssystem in den Browser verlagert. Unterschiedliche Browser-Implementierungen und -Versionen stellen Webseiten auf unterschiedliche Weise dar, was es bei der Anwendungsentwicklung zu berücksichtigen gilt. Weiterere Vorteile von Web-Anwendungen über die Betriebssystemunabhängigkeit hinaus sind

die sowohl aus Benutzer- als auch aus Betreibersicht vereinfachte Verwaltung und der Betrieb der Anwendung, die zentral auf den Rechnern des Betreibers installiert ist. Aktualisierungen sind sofort bei der nächsten Verwendung verfügbar, sofern eine Internetverbindung besteht. Eine aufwändige Verteilung und Installation auf den mobilen Geräten ist dabei nicht nötig. Insofern ist der folgende Abschnitt nicht spezifisch für das Betriebssystem Linux, spielt aber in diesem Umfeld dennoch eine zunehmend wichtige Rolle. Allerdings werden nur die Konzepte erklärt, nicht aber konkrete Beispiele gezeigt.

Der Nachteil von Web-Anwendungen ist die teilweise mangelnde, schwierige oder gar unmögliche Integration in das Erscheinungs- und Bedienungskonzept des Gerätes (engl.: Look and Feel). Ausserdem kann auf die spezifischen Besonderheiten der Hardware des Gerätes nur bedingt eingegangen werden.

Obwohl viele Darstellungsformen und die dafür benötigten technischen Voraussetzungen inzwischen auch auf mobilen Endgeräten verfügbar sind, müssen Webseiten für diese Browser vorbereitet werden.

6.7.1 WebKit

Mittlerweile hat eine Konsolidierung im Bereich der Implementierung von Komponenten zur Aufbereitung von Webseiten (engl.: Rendering Engines) eingesetzt und sich das quelloffene *WebKit* Projekt als de-facto Standard durchgesetzt. Diese Tatsache erleichtert die Entwicklung plattformübergreifender Webanwendungen erheblich. Das WebKit-Projekt erreicht man unter der Adresse http://webkit.org/.

6.7.2 HTML5

Der vom *World Wide Web Consortium* (*W3C*) definierte Standard *HTML5* enthält einige Neuerungen, die das mobile Arbeiten mit Web-Anwendungen erleichtern und teilweise erst praktikabel machen.

6.7.2.1 Lokale Persistenz

Für den Wunsch, auch aus Web-Anwendungen heraus Anwendungsdaten lokal zu speichern, also auf dem Rechner, auf dem der Web-Browser läuft, gibt es mehrere gute Gründe. Beim Einsatz von Web-Anwendungen war lange Zeit eine ständige Verbindung zum Internet Voraussetzung, wenn keine Zusatzsoftware, wie z. B. so genannte Browser-Plug-Ins installiert werden sollte oder konnte. Gerade bei mobilen Geräten kann aber eine ständige Verbindung ins Netz nicht ohne Weiteres sichergestellt werden. Wenn aber die Daten auf dem Gerät verfügbar sind, ist eine ständige Netzverbindung keine notwendige Voraussetzung.

Darüber hinaus ist das Abrufen der kompletten Seitendaten bei jedem Zugriff aufwändig und kann, abhängig von der Geschwindigkeit der Verbindung, relativ

lange dauern. Daten können von einem lokalen Speichermedium üblicherweise deutlich schneller geladen werden.

Der Standard *HTML5* führt nun die Möglichkeit ein, Daten auf dem Rechner abzulegen, auf dem der Browser ausgeführt wird und zwar in grösserem Umfang und mit einem anderen Ziel, als das bisher schon mit der so genannten *Cookie*-Technik möglich war. Cookies erlauben es, kleinere Datenmengen lokal zu speichern. Diese Daten werden bei jedem Zugriff auf die Webseite, von der sie stammen, an den Server zurückgeschickt und können von diesem ausgewertet werden. Mit der *Web Storage* genannten Spezifikation wird das Konzept einer Datenbank eingeführt, in der auch grössere Datenmengen persistiert werden können. Das macht es zum einen möglich, auch dynamische Inhalte lokal zwischenzuspeichern und vorzuhalten, um beim erneuten Bedarf schnellere Zugriffszeiten zu erzielen (engl.: Cache), zum anderen wird keine permanente Netzverbindung mehr benötigt, und es kann in gewissem Umfang autark gearbeitet werden. Für statische Inhalte gibt es das Konzept eines lokalen Cache schon länger. Auf die Datenbank kann mit Hilfe von JavaScript zugegriffen werden.

6.7.2.2 Netzunabhängiges Arbeiten

Damit eine ständige Verbindung zum Internet keine Voraussetzung mehr für den Einsatz von Web-Anwendungen ist, führt HTML5 das Konzept netzunabhängiger Web-Anwendungen (engl.: Offline Web Applications) ein.

Damit der Browser dynamische Inhalte zwischenspeichern kann, werden sämtliche zur Anwendung gehörende Dateien in einem so genannten *Cache-Manifest* aufgelistet. Diese Datei muss dem Browser über das HTML5 Attribut `manifest` bekanntgegeben werden, z. B.:

```
<html manifest="offline-manifest.mf">
```

Die Deklaration der Cache-Manifest-Datei muss dabei in jeder HTML-Seite erfolgen, die zu der Web-Anwendung gehört, damit die Information an den Browser ausgeliefert wird, unabhängig davon, welche Seite aufgerufen wird. Dabei wird immer die selbe Datei referenziert.

Die Dateipfade in der Datei müssen relativ zur Cache-Manifest-Datei angegeben werden. Browser-Implementierungen, die HTML5 unterstützen, speichern die aufgelisteten Dateien lokal. Sobald der Browser eine neue Version des Cache-Manifest erhält, muss er die gespeicherten Dateien aktualisieren. Dabei ist es egal, was sich in der Datei geändert hat, es reicht aus, einen Kommentar zu aktualisieren. Der Name der Manifest-Datei kann frei gewählt werden.

Für das Cache-Manifest sind drei mögliche Sektionen vordefiniert:

CACHE: Dateien in dieser Sektion werden vom Browser in den Cache geladen. Das ist auch der Standard, wenn die beiden anderen Sektionen fehlen. In diesem Fall kann der Sektionstitel entfallen.

NETWORK: In dieser Sektion werden die Dateien aufgelistet, die nicht im Cache gehalten werden sollen, sondern in jedem Fall aus dem Netz geladen werden müssen. Wird im Offline-Fall auf eine Ressource in dieser Sektion zugegriffen, zeigt der Browser eine Fehlermeldung an.

FALLBACK: Mit dieser Sektion kann darauf reagiert werden, wenn im Offline-Fall auf eine Datei zugegriffen wird, die sich aus einer Vielzahl möglicher Gründe nicht im Cache befindet. Es wird dann statt der eigentlich angeforderten Datei der angegebene Ersatz-Inhalt angezeigt.

Die Fallback-Sektion besteht aus Mustern bzw. Schablonen für Web-Adressen (URLs) und den Seiten, deren Inhalt angezeigt werden soll, falls sich eine Seite nicht im Cache befindet.

Der MIME-Typ spezifiziert für jede Datei, die vom Browser geladen werden soll, wie die Datei zu verarbeiten ist, bzw auf welche Art sie übertragen wird. Für die Datei, die das Cache-Manifest enthält, muss `text/cache-manifest` als MIME-Typ deklariert werden. Das muss entsprechend auf dem Web-Server, der die Anwendung bereitstellt, konfiguriert werden.

Der Zugriff auf Daten in der lokalen Datenbank kann nur von Webseiten mit derselben Herkunft (engl.: Origin) erfolgen. Auf diese Weise wird verhindert, dass Anwendungen von unterschiedlichen Webseiten gegenseitig Daten lesen oder schreiben können, was die Datensicherheit kompromittieren würde. Die Datenbank selbst ist als eine Menge an Schlüssel-Wert-Paaren aufgebaut. Schlüssel bestehen aus Zeichenketten (engl.: String).

Die Beschreibung zum netzunabhängigen Arbeiten in diesem Abschnitt betrifft nur den statischen Anteil einer Web-Anwendung, also Dateien, die unverändert vom Webserver zum Browser geschickt werden. Wenn eine Web-Anwendung selbst Daten speichern muss, ist es die Aufgabe der Anwendung, dies abhängig vom Online-Status durchzuführen (Abschn. 6.7.2.1).

6.7.2.3 Ortsbestimmung (Geolocation)

Geolocation, also die Bestimmung der physischen Position eines Anwenders, genauer die des eingesetzten Gerätes, wird nicht im HTML5 Standard definiert, sondern in einer parallelen Arbeitsgruppe des W3C, die sich mit speziell dieser Thematik auseinandersetzt.

Um die Position eines mobilen Gerätes zu bestimmen, gibt es verschiedene Methoden, die, wenn sie kombiniert werden, die Präzision erhöhen. Die beste Möglichkeit ist der Einsatz eines Empfängers für das *Globale Positionsbestimmungssystem* (*NAVSTAR GPS*) (engl.: Global Positioning System (GPS)), bei dem mit Hilfe von Satelliten in der Erdumlaufbahn die Position sehr genau bestimmt werden kann. Weitere Möglichkeiten sind die Auswertung der Signalstärke von Mobilfunkmasten, die verwendete IP-Adresse und die Einbeziehung von Lokationsinformationen und der Signalstärke empfangener Wireless LAN Netzwerke.

6.7.3 PhoneGap

Das Projekt *PhoneGap* (http://www.phonegap.com) möchte die Unterschiede bei der Umsetzung mobiler Anwendungen für diverse Anbieter überbrücken und dabei trotzdem den Bedienungskomfort des jeweils eingesetzten Gerätes beibehalten. Es wird eine einheitliche Programmierschnittstelle definiert und diese für unterschiedliche Geräte und Plattformen so implementiert, dass die Anwendung sich aus Benutzersicht in das Bedienkonzept des Gerätes einfügt. Ziel ist es, Anwendungen mit offenen Standards wie HTML, CSS und JavaScript die Möglichkeit zu geben auf gerätespezifische Funktionalität zugreifen zu können, dabei aber trotzdem plattformunabhängig zu bleiben.

Unterstützte Funktionalität schliesst Beschleunigungssensor (Accelerometer), Kamera, Ortsbestimmung, Netzwerk, Benutzerbenachrichtigungen sowie den Zugriff auf gerätespezifische Funktionalität und die Reaktion auf Ereignisse (engl.: Events) der verwendeten Plattform ein. Bei PhoneGap werden Anwendungen zwar mit Web-Standards wir HTML und JavaScript entwickelt, jedoch handelt es sich bei dem Ergebnis nicht um herkömmliche Web-Anwendungen. Abbildung 6.9 veranschaulicht die Vorgehensweise. Die entwickelte Anwendung wird in die Erstellungsumgebung von PhoneGap geladen und dort in die Installationsformate der unterstützten Plattformen gepackt. Dabei steuert PhoneGap die plattformabhängigen Bestandteile bei und überbrückt somit die Unterschiede zwischen den Plattformen. Das Ergebnis sind plattformabhängige Anwendungen, die auf der jeweiligen Plattform installiert werden können.

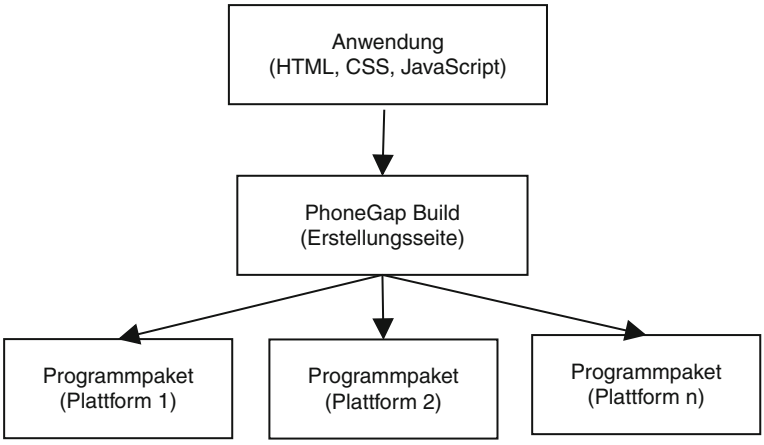

Abb. 6.9 Die Funktionsweise von PhoneGap

6.7.4 Wireless Application Protocol (WAP)

Das *Wireless Application Protocol* war ein erster Versuch, Inhalte aus dem World Wide Web speziell für mobile Geräte mit kleiner Darstellungsfläche aufzubereiten und zu optimieren. Dieses Protokoll hat sich für die mobile Welt nicht durchgesetzt.

6.8 Globalisierung

Unter dem Stichwort *Globalisierung* wird die Funktionalität zusammengefasst, die es ermöglicht, Software mit unterschiedlichen menschlichen Sprachen und kulturellen Normen einsetzen zu können. Globalisierung wird auch mit *G11N* abgekürzt, wobei *11* für die elf Buchstaben steht, die sich im englischen Wort *Globalization* zwischen Anfangs- und Endbuchstaben befinden.

Im Umfeld von Globalisierung fallen oft Schlagworte wie *Internationalisierung* und *Lokalisierung*.

6.8.1 Internationalisierung

Unter *Internationalisierung* (*engl. Internationalization – I18N*) sei in diesem Zusammenhang verstanden, dass eine Anwendung in unterschiedlichen Sprachen bedient werden kann. Dazu ist es u.a. nötig, dass die auf der Benutzeroberfläche angezeigten Texte aus dem Programm-Quelltext extrahiert und in die unterstützten Sprachen übersetzt werden. Selbst wenn ein Programm nur in einer Sprache angeboten werden soll, ist es sinnvoll, diese Trennung zwischen Darstellung und Programmcode vorzunehmen. Zum einen wird dadurch eine möglicherweise zu einem späteren Zeitpunkt gewünschte Übersetzung vereinfacht und zum anderen dient es der Strukturierung des Quellcodes, indem die Anzeige klar von der Programmlogik getrennt ist. Abhängig von der Auswahl des Benutzers werden dann die Texte in der Sprache der Wahl angezeigt.

Der für die Anzeige eines Textes benötigte Platz variiert von Sprache zu Sprache erheblich. Deutsche Texte sind in der Regel länger und daher weniger platzsparend als englische. Asiatische Übersetzungen können dagegen mit wenigen Schriftzeichen recht kurz ausfallen. Bei manchen Sprachen, z. B. aus dem arabischen Raum, ändert sich die Leserichtung von *links nach rechts* nach *rechts nach links*. In solchen Fällen muss sich das gesamte Layout der Oberfläche nach der Leserichtung ausrichten. Diese Umstände sind weitere wichtige Gründe, warum sich der Aufbau einer Benutzeroberfläche nicht an festen Grössenverhältnissen orientieren sollte, sondern sich möglichst dynamisch an die gegebenen Anforderungen anpassen sollte.

6.8.1.1 GNU gettext

Auf der Webseite http://www.gnu.org/software/gettext nimmt sich das GNU Projekt mit *gettext* der Internationalisierung von C-Programmen, aber auch anderer

Programmiersprachen an. Texte eines Programms, die übersetzt und in mehreren Sprachen angeboten werden sollen, werden in so genannten PO Dateien abgelegt, wobei PO für *Portable Object* steht. Dabei wird für jede unterstützte Sprache eine eigene PO Datei angelegt.

Um die Funktionalität von gettext nutzen zu können, werden die C-Quelldateien oder zentrale Header-Dateien um folgenden Code erweitert.

```
#include <libintl.h>
#include <locale.h>
#define _(String) gettext (String)
#define gettext_noop(String) String
#define N_(String) gettext_noop (String)
```

Die Header-Datei `libintl.h` stellt die Deklarationen für den Aufruf von gettext bereit und `locale.h` enthält Deklarationen zu möglichen Spracheinstellungen. Die drei darauf folgenden Makros erleichtern die Verwendung von gettext und sollen den Quellcode lesbarer machen.

Statt Texte fest in den Quelltext zu schreiben, wird der Inhalt über die Funktion `gettext` aus einem externen Katalog geladen. Als Parameter für `gettext` dient der auszugebende Text in der Standardsprache, in der das Programm entwickelt wird. Dieser Text stellt eine eindeutige Kennung dar, mit Hilfe derer `gettext` den Text in der zur Laufzeit eingestellten Sprache einliest. Insbesondere wenn viele Textausgaben und damit viele Aufrufe von `gettext` vorkommen, hat es sich bewährt, den Aufruf von `gettext` mit dem Makro `_(String)` abzukürzen. Im Quellcode steht dann statt `gettext("Originaltext")` der Aufruf `_("Originaltext")`. Beim Verfassen der Originaltexte ist es sinnvoll, nur Zeichen aus dem ASCII Zeichensatz zu verwenden, um Komplikationen bei der Verarbeitung mit den Werkzeugen zur Internationalisierung zu vermeiden.

Es gibt Fälle, in denen der zu übersetzende Text im Quellcode nicht direkt durch `gettext` ersetzt werden kann, da an der konkreten Stelle kein Funktionsaufruf möglich ist, sondern ausschliesslich ein statischer Text. Für solche Fälle wurde das Makro `gettext_noop(String)` definiert. Es dient dazu, den enthaltenen Text für die Übersetzung zu markieren. Aus den gleichen Gründen wie bei `gettext` kommt noch das Makro N_(String) als Kurzform dazu. Die Idee bei diesem Ansatz ist es, zunächst nur die Kennung des Texts anzugeben und erst zu dem Zeitpunkt, an dem der Text tatsächlich ausgegeben wird, mittels `gettext` die übersetzte Version aus dem Katalog auszulesen.

Bevor die Bibliothek mit der Internationalisierungsfunktionalität im Programm verwendet werden kann, muss sie entsprechend initialisiert werden. Das geschieht üblicherweise genau einmal beim Start des Programms.

```
setlocale(LC_ALL, "");
bindtextdomain("verbrauch", ".");
textdomain("verbrauch");
```

Mit dem Aufruf von `setlocale` wird die Umgebung darauf vorbereitet, mit mehreren Sprachen umzugehen. Der Aufruf von `textdomain` sorgt dafür, dass

übersetzte Texte nur für die angegebene Domäne gesucht werden und entspricht damit einer Art Namensraum. `bindtextdomain` teilt dem System mit, in welchem Verzeichnis nach Texten der angegebenen Domäne gesucht werden soll. In obigem Beispiel sind die Parameter für Domäne und Verzeichnispfad zum besseren Verständnis mit festen Werten kodiert. In konkreten Entwicklungsprojekten werden die Werte für die Parameter üblicherweise mit den Makros `PACKAGE` und `LOCALEDIR` gesetzt, die in der zentralen Konfigurations-Header-Datei `config.h` definiert werden. Das Programmbeispiel aus Abschn. 6.2.1.1 sieht entsprechend für die Internationalisierung vorbereitet folgendermassen aus.

```
#include <stdio.h>
#include <stdlib.h>
#include "durchschnitt.h"
#include <libintl.h>
#include <locale.h>
#define _(String) gettext (String)
#define gettext_noop(String) String
#define N_(String) gettext_noop (String)

int main(int argc, char *argv[])
{
  setlocale (LC_ALL, "");
  bindtextdomain ("verbrauch", ".");
  textdomain ("verbrauch");

  printf(_("Verbrauch pro 100 Kilometer: %f\n"),
    durchschnitt(atoi(argv[1]), atoi(argv[2])) * 100);
  return 0;
}
```

Durch die oben beschriebene Vorgehensweise werden alle zu übersetzenden Texte entweder mit `gettext` oder `gettext_noop(String)` im Quellcode gekennzeichnet. Die so markierten Texte können nun mit dem Werkzeug `xgettext` aus dem Quellcode ausgelesen werden. Damit die mit dem Makro `#define _(String) gettext (String)` markierten Stellen gefunden und extrahiert werden, muss beim Aufruf der zusätzliche Parameter `-k` angegeben werden.

```
xgettext -k_ verbrauch.c
```

Das Ergebnis ist eine Datei `messages.po`, wenn nicht mit dem Parameter `-o` ein anderer Dateiname angegeben wurde.

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE
package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
```

```
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2011-03-20 14:43+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: verbrauch.c:16
#, c-format
msgid "Verbrauch pro 100 Kilometer: %f\n"
msgstr ""
```

In dieser Datei sollten die eingefügten Platzhalter wie `SOME DESCRIPTIVE TITLE`, `YEAR`, `PACKAGE` usw. durch sinnvolle Werte ersetzt werden. Die ersten Zeilen enthalten Kommentare und den Meta-Kommentar `#, fuzzy`, zu erkennen an der Zeichenfolge `#,` am Zeilenanfang. Danach folgt eine Liste mit `msgid` und `msgstr` Paaren. Das erste Paar hat einen leeren Eintrag als Wert für `msgid` und mehrere Zeilen beschreibenden Text zur vorliegenden Datei. Die Unterteilung in mehrere Zeilen dient nur der besseren Lesbarkeit, bei der weiteren Verarbeitung werden die Zeilen zu einer einzigen langen Zeile zusammengefügt. Auch in diesem Abschnitt sollten die Platzhalter mit sinnvollen Werten versehen werden, wobei der Wert für `CHARSET` gesetzt werden muss. Über diesen Wert wird spezifiziert, in welchem Kodierungsformat für Textzeichen die Datei gespeichert ist. Am unproblematischsten ist die Verwendung von UTF-8, auf jeden Fall muss aber sichergestellt werden, dass die Datei immer im angegebenen Kodierungsformat abgelegt wird.

Der Meta-Kommentar enthält zusätzliche Informationen zum darauffolgenden Eintrag. Im Fall von `fuzzy` wird damit signalisiert, dass der Eintrag noch nicht vollständig ist und noch bearbeitet werden soll. Sobald dies nicht mehr der Fall ist, kann die Zeile mit dem Meta-Kommentar entfernt werden. Des Weiteren enthält die Datei für jeden mittels `gettext` markierten und extrahierten Text einen Eintrag nach folgendem Schema.

```
#: verbrauch.c:50
msgid "Originaltext"
msgstr ""
```

Die erste Zeile gibt den Dateinamen und die Zeile des Quellcodes an, aus der der Text stammt. Mit `msgid` wird der zu übersetzende Text identifiziert und `msgstr` enthält einen leeren Platzhalter für die Übersetzung. Diese Datei dient als Vorlage für die Übersetzungen, deshalb sollte die Dateiendung in `.pot` umbenannt werden, wobei die Abkürzung für *Portable Object Template* steht. Für jede unterstützte Spra-

che wird eine Kopie dieser Datei als `.po` Datei erstellt, die die entsprechenden Übersetzungen enthält.

Das Werkzeug `xgettext` überschreibt vorhandene Dateien. Wenn im Quellcode zusätzliche Texte hinzukommen, können die entsprechenden Einträge in den `.po` und `.pot` Dateien entweder manuell oder mit Hilfe des Werkzeugs `msgmerge` eingefügt werden. `msgmerge` erhält als Parameter eine existierende Datei mit enthaltenen Übersetzungen und eine mit `xgettext` neu erzeugte Datei mit den aktuellen Text-Kennungen. Die Ergebnisdatei wird mit dem Parameter `-o` spezifiziert, sonst wird das Ergebnis über die Standardausgabe präsentiert. Die Ergebnisdatei enthält daraufhin alle bereits existierenden Text-Kennungen und deren Werte sowie die neu hinzugekommenen Text-Kennungen.

Sobald die übersetzten PO-Dateien vorliegen, werden diese mit dem Werkzeug `msgfmt` in eine maschinenlesbare Form mit der Dateiendung `.mo` überführt. Die Abkürzung *MO* steht dabei für *Machine Object*. Für jede unterstützte Sprache gibt es eine MO-Datei. Die Verzeichnisstruktur sieht ein Unterverzeichnis für jede Sprache vor, wobei der Verzeichnisname dem Sprachkürzel entspricht, also z. B. `de` für Deutsch. In jedem dieser Unterverzeichnisse wird ein weiteres Unterverzeichnis `LC_MESSAGES` erwartet und in diesem Verzeichnis sucht die Internationalisierungsbibliothek nach den MO-Dateien. Im Beispiel oben wurde `verbrauch` als Wert für die Domäne angegeben, insofern sucht das Programm nach einer Datei `verbrauch.mo` für die zur Laufzeit verwendete Sprache. Im Fall von Deutsch ist das die Datei `de/LC_MESSAGES/verbrauch.mo`.

Bibliotheken

Die bisherige Beschreibung zur Internationalisierung gilt für eigenständig lauffähige Programme. Die Vorgehensweise, um eine Funktionsbibliothek zu internationalisieren, weicht geringfügig davon ab.

Die Aufrufe von `setlocale` und `textdomain` entfallen, der Programmcode der Bibliothek verlässt sich darauf, dass ein Programm, das die Funktionalität der Bibliothek nutzt, die Umgebung entsprechend initialisiert hat. Bei der Initialisierung der Bibliothek bleibt lediglich der Aufruf von `bindtextdomain` übrig, damit die zur Bibliothek gehörenden Textkataloge gefunden werden. Da die Bibliothek mit diesem Aufruf ihre eigene Domäne für die Suche nach Texten bekanntgibt, muss auch statt `gettext` die Funktion `dgettext` verwendet werden, die als zusätzlichen Parameter den Domänennamen benötigt. Damit ändert sich auch das Makro für die Kurzform entsprechend.

```
#define _(String) dgettext (PACKAGE, String)
```

6.8.1.2 Java Internationalisierung

Beim Entwurf der Java Plattform (Abschn. 6.2.2) wurde Internationalisierung von vornherein mit einbezogen. Dementsprechend fortgeschritten ist die Unterstützung bei der Softwareentwicklung. Beispielsweise werden Zeichenketten in der Java

Plattform immer als Unicode (Abschn. 3.5.1.2) repräsentiert. Die Sprachdefinition beinhaltet u.a. eine Klasse `Locale`, mit der Sprache, Land und ggf. regionale Ausprägungen repräsentiert werden. Klassen zur Formatierung von Zahlen, Datums- und Währungsangaben wiederum machen von dieser Information gebrauch und stellen die gewünschten Daten passend formatiert dar.

Zu übersetzende Texte werden in so genannte *Properties*-Dateien mit der Dateiendung `.properties` extrahiert. Für jede unterstützte Sprache bzw. jedes Land wird eine eigene Properties-Datei angelegt, wobei Sprache, Land und ggf. regionale Besonderheiten über die entsprechenden Abkürzungen im Dateinamen angegeben werden. Für eine Properties-Datei, die US-amerikanisches Englisch enthält, endet der Dateiname beispielsweise mit `_en_US.properties`.

Bei Properties-Dateien handelt es sich um Textdateien, mit einer Liste von Schlüsselworten, denen jeweils ein Text-Wert zugeordnet ist. Jeder zu übersetzende Text wird mit einem eindeutigen Schlüsselwort gekennzeichnet. Für das in Abschn. 6.2.2.3 vorgestellte Beispielprogramm sieht eine passende Properties-Datei mit dem Basisnamen `verbrauch` für Deutschland `verbrauch_de_DE.properties` folgendermassen aus.

```
VERBRAUCH=Verbrauch pro 100 Kilometer:
```

Eine Datei ohne Land- bzw. Sprachabkürzung im Dateinamen enthält die Werte für die Standardsprache. Die in dieser Datei enthaltenen Texte werden immer dann angezeigt, wenn für die gewünschte Sprach-/Land-Kombination keine übersetzten Texte vorliegen und somit keine Properties-Datei existiert. Eine solche Properties-Datei, die im Dateinamen nur den Basisnamen enthält, sollte immer angelegt werden, damit auf sie zurückgegriffen werden kann, wenn die gewünschte Übersetzung nicht existiert.

Um übersetzte Texte aus einer Properties-Datei zu lesen, sieht die Java Plattform die Klasse `ResourceBundle` vor. Um ein Objekt dieser Klasse zu erhalten, steht die statische Methode `getBundle` zur Verfügung. Als Parameter wird der Basisname der Properties-Dateien übergeben, also der gemeinsame Name ohne Sprach- und Landabkürzungen. Dabei wird die eingestellte Standard-Locale ermittelt und die Texte aus der entsprechenden Properties-Datei ausgelesen. Eine weitere Implementierung der Methode `getBundle` erwartet in einem zweiten Parameter ein Objekt vom Type `Locale`, um die Werte für eine bestimmte Locale zu laden. Eine internationalisierte Version des Programms besteht aus folgendem Quelltext.

```
package com.springer.durchschnitt.i18n;

import com.springer.durchschnitt.Durchschnitt;
import com.springer.durchschnitt.DurchschnittStandard;
import java.util.ResourceBundle;

public class Verbrauch
{
  public static void main(String args[])
```

```
  {
    Durchschnitt durchschnitt = new DurchschnittStandard();

    ResourceBundle ressourcen =
        ResourceBundle.getBundle("verbrauch");

    System.out.println(ressourcen.getString("VERBRAUCH")
      + durchschnitt.durchschnitt(Integer.parseInt(args[0]),
                                  Integer.parseInt(args[1]))
      * 100);
  }
}
```

Damit die Java Umgebung die Properties-Dateien findet, müssen sich diese wie ausführbarer Code auch im Suchpfad der Umgebung befinden, der üblicherweise über die Umgebungsvariable `CLASSPATH` definiert wird.

Textzeichen, die nicht mit dem ASCII Zeichensatz dargestellt werden können, werden in den Properties-Dateien mit ihrer Unicode-Kennung (Abschn. 3.5.1.2) abgelegt. Um die Eingabe der Übersetzungen möglichst unkompliziert zu gestalten und die Übersetzer nicht mit Unicode-Werten zu belasten, bietet die Java Entwicklungsumgebung das Werkzeug `native2ascii` an. Das ermöglicht es, Texte im Kodierungsformat der Entwicklungsumgebung abzuspeichern. `native2ascii` liest die Textdatei ein und transformiert alle Zeichen ausserhalb des ASCII Zeichensatzes in die entsprechende Unicode-Kennung. Der erste Parameter ist der Name der einzulesenden Datei und der zweite Parameter der Name der Ausgabedatei. Fehlt dieser, erfolgt die Ausgabe über die Standardausgabe.

6.8.1.3 Pango

Pango ist eine Programmbibliothek zur Darstellung von Text. Ein Schwerpunkt wird dabei auf Internationalisierung gelegt. Beispielsweise unterstützt Pango von vornherein bidirektionalen Text (BiDi), d.h. dass z. B. für arabische Sprachen und Hebräisch die Leserichtung und Darstellung statt von links nach rechts umgekehrt von rechts nach links erfolgt. Die Projektseite im Web findet sich unter http://www.pango.org. Der Bedarf für eine Bibliothek wie Pango entstand aus dem GTK+ Projekt, um GTK+ basierte Oberflächen wie z. B. GNOME in internationalisierten Versionen anbieten zu können.

Für Zeichenketten verwendet UTF-8 kodierte Unicode-Zeichen (Abschn. 3.5.1.2).

Um unterschiedliche Sprachen und Sprachvarianten zu unterstützen, arbeitet Pango mit so genanntem *Tagging*, wobei der anzuzeigende Text mit Meta-Information angereichert wird. Anhand dieser Meta-Information, wie z. B. Sprache, Schriftgrösse und -farbe stellt Pango den Text entsprechend aufbereitet dar. Die Attribute der Meta-Information werden in eigenen Strukturen separat von dem anzuzeigenden Text gehalten und beziehen sich jeweils auf einen bestimmten Abschnitt der Zeichenkette. Insbesondere wenn die Anzahl der Attribute wächst und spätestens wenn Übersetzungen der Texte vorliegen, ist diese Vorgehensweise nicht

mehr praktikabel. Daher bietet Pango an, die Meta-Information mit Hilfe von Tags und Attributen in den Text einzubetten. Die Syntax der Tags entspricht der von XML, d.h. die Tag-Namen werden in spitze Klammern eingefasst, wobei Gross- und Kleinschreibung beachtet wird. Beispielsweise bezeichnet die folgende Zeile den Beginn eines deutschen Textabschnitts.

```
<span lang="de_DE">
```

Der Wert für das Attribut `lang` wird als *Locale* bezeichnet. Die Locale legt über definierte Abkürzungen die Sprache sowie das Land fest. Beispielsweise wird kanadisches Französisch mit `fr_CA` beschrieben. Darüber hinaus können abhängig von der Implementierung noch regionale Spezifizierungen oder der verwendete Zeichensatz angegeben werden.

6.8.2 Lokalisierung

Unter *Lokalisierung* (*engl.: Localization – L10N*) versteht man die Berücksichtigung regionaler Konventionen, also z. B. die Einbeziehung der Darstellung von Formaten von Zeiten, Datumsangaben, Zahlen etc, abhängig von den Präferenzen des Benutzers, die sich in den meisten Fällen an den regional gebräuchlichen Normen orientieren. Beispielsweise wird ein Datum in Deutschland üblicherweise in der Notation `TT.MM.JJJJ` dargestellt, in den Vereinigten Staaten mit `MM/TT/JJJJ`, während die ISO-Norm `JJJJ-MM-TT` vorsieht. Ähnliches gilt für die Trennzeichen in Zahlen und Zeitangaben.

6.9 Barrierefreiheit

Ein wichtiger Aspekt bei der Entwicklung von Produkten und Lösungen, also auch in der Softwareentwicklung ist die *Barrierefreiheit* (*engl.: Accessibility*), die es Menschen mit körperlichen Einschränkungen erlaubt, die Ergebnisse zu benutzen. Beispielsweise kann in vielen Fällen als Alternative zum Mausklick eine Tastenkombination für das Auslösen einer Aktion angeboten werden. Derartige Konzepte können auch für nichtbehinderte Anwender hilfreich sein, da z. B. die Bedienung eines Programms über Tastenkombinationen schneller sein kann als mit der Maussteuerung. Ein weiteres Beispiel ist das Hinterlegen von Metainformationen zu den Bedienelementen auf der Benutzeroberfläche, die von entsprechender Software aus- und vorgelesen werden kann, um blinden oder sehbehinderten Personen die Navigation zu ermöglichen. Gehörlosen Menschen hilft es, wenn Video-Inhalte mit Untertiteln versehen sind und der Inhalt von Audio-Dateien textuell bzw. visuell abrufbar ist.

Leider steht die Barrierefreiheit bei vielen Entwicklern nicht an erster Stelle. Ein grosser Vorteil quelloffener Software ist es in diesem Zusammenhang, dass entsprechende Funktionalität auch noch nach deren Veröffentlichung von der Entwickler-

gemeinschaft eingebracht, bzw. die Software bei Bedarf für den konkreten Anwendungsfall entsprechend ausgerüstet werden kann. Gerade das Pervasive Linux mit neuen Möglichkeiten auf den Benutzer zu reagieren wie z. B. Bewegungssensoren, bietet viele Chancen, den Grad der Barrierefreiheit zu erhöhen.

Um bessere Lesbarkeit zu erreichen, werden für sehbehinderte Menschen oft besondere Kontrast- und Farbschemata und grössere Schriftarten als gewöhnlich eingesetzt. Das setzt voraus, dass ein Programm mit den geänderten Grössenverhältnissen umgehen und den Inhalt in jedem Fall vollständig anzeigen kann. Dies zu berücksichtigen ist ein weiterer Aspekt der eingangs in Abschn. 1.1.2 bereits angesprochenen Flexibilität in Bezug auf Grössenverhältnisse und Auflösungen der Anzeigegeräte. Das gilt natürlich sowohl für eigenständige Programme, die direkt auf dem Gerät ausgeführt werden, als auch für Web-Anwendungen.

Für Web-Anwendungen sieht HTML5 (Abschn. 6.7.2) die Barrierefreiheit als integrierten Sprachbestandteil vor. Gerade bei deklarativen Oberflächen wie HTML kann der Inhalt sehr gut von der Darstellung getrennt werden. Dabei lassen sich schon durch einfache strukturelle Anpassungen erhebliche Vorteile für barrierefreie Repräsentationen erreichen, z. B. indem der Inhalt bzw. die Hauptfunktion einer Seite oder Oberfläche vor deren Navigation steht. Auf diese Weise können alternative Ausgabemedien, wie z. B. die Sprachausgabe dessen, was sonst auf dem Bildschirm angezeigt wird, den Inhalt sequenziell auslesen und wiedergeben. Somit wird der Hauptinhalt vor den Navigationsmöglichkeiten wiedergegeben. Die bessere Strukturierung und zusätzlichen Meta-Informationen führen darüber hinaus auch dazu, dass Internet-Suchmaschinen eine Seite bzw. Anwendung besser kategorisieren und somit gezielter in Suchergebnissen anzeigen können.

Literaturverzeichnis

1. Gough BJ (2004) An introduction to GCC. Network Theory Ltd
2. Becker A, Pant M (2010) Android 2: Grundlagen und Programmierung. dpunkt.verlag
3. Wietzke J, Tien TM (2005) Automotive Embedded Systeme: Effizientes Framework – Vom Design zur Implementierung. Springer
4. Calcote J (2010) Autotools: A practioner's guide to GNU autoconf, automake, and libtool. No Starch Press
5. Yaghmour K (2003) Building embedded linux systems. O'Reilly
6. Blanchette J, Summerfield M (2008) C++ GUI programming with Qt 4. Prentice Hall
7. Goldfarb C, Prescod P (2000) Das XML-Handbuch. Anwendungen, Produkte, Technologien. Addison-Wesley
8. Zwick C, Schmitz B, Kühl K (2005) Designing for small screens. AVA Publishing
9. Stroustrup B (2000) Die C++ Programmiersprache. Professionelle Programmierung. Addison Wesley
10. Stöhr F (2007) Die GNU autotools. C&L Computer und Literaturverlag
11. Pilgrim M (2004) Dive into Python, Apress
12. Schröder J, Gockel T, Dillmann R (2009) Embedded linux. Springer
13. Abbott D (2008) Embedded linux development using eclipse. Newnes
14. Hallinan C (2006) Embedded linux primer. Prentice Hall
15. Venkateswaran S (2008) Essential linux device drivers. Prentice Hall International
16. Fornari (2009) Funambol mobile open source. Packt Publishing
17. Vaughan G, Elliston B, Tromey T (2000) GNU autoconf, automake, and libtool. New Riders Publishing
18. Luszczak A (2004) Grundkurs Socketprogrammierung mit C unter Linux. Vieweg+Teubner
19. Münz S, Gull C (2010) HTML 5 Handbuch. Franzis
20. Pilgrim M (2010) HTML5: Up and running. O'Reilly
21. Darwin IF (2002) Java Kochbuch. O'Reilly
22. Wenz C (2010) JavaScript. Galileo Press
23. Gulbins J, Obermayr K, Snoopy (2003) Linux. Springer
24. Kofler M (2006) Linux. Addison Wesley
25. Corbet J, Rubini A, Kroah-Hartman G (2005) Linux device drivers. O'Reilly
26. Wolfinger C, Gulbins J, Hammer C (2004) Linux Systemadministration. Springer
27. Herold H (2004) Linux/Unix Systemprogrammierung. Addison Wesley
28. Quade J, Kunst E-K (2006) Linux-Treiber entwickeln, dpunkt.verlag
29. Roth J (2005) Mobile computing. dpunkt.verlag
30. Fling B (2009) Mobile design and development. O'Reilly
31. Korff A (2008) Modellierung von eingebetteten Systemen mit UML und SysML
32. Schneeweiss R (2007) Moderne C++ Programmierung. Springer
33. Schellong H (2005) Moderne C-Programmierung. Springer

C. Czernohous, *Pervasive Linux*, X.systems.press,
DOI 10.1007/978-3-540-68426-8, © Springer-Verlag Berlin Heidelberg 2012

34. McAffer J, VanderLei P, Archer S (2010) OSGi and equinox: Creating highly modular Java systems. Addison-Wesley Professional
35. Hall RS, Pauls K, McCulloch S, Savage D (2011) OSGi in action. Manning
36. Kernighan B, Ritchie D (1990) Programmieren in C. ANSI C. Hanser
37. Barr M, Massa A (2006) Programming embedded systems. O'Reilly
38. Ziegler J (2002) Programmieren lernen mit Perl. Springer
39. Wall L, Christiansen T, Orwant J (2001) Programmieren mit Perl. O'Reilly
40. Firtman M (2010) Programming the mobile web. O'Reilly
41. Dalheimer MK, Welsh M (2005) Running linux. O'Reilly
42. Osterhage W (2010) Sicher & Mobil. Springer
43. Bless R, Mink S, Blaß E-O, Conrad M, Hof H-J, Kutzner K, Schöller M (2005) Sichere Netzwerkkommunikation. Springer
44. Liggesmeyer P, Rombach D (2005) Software-Engineering eingebetteter Systeme. Spektrum Akademischer Verlag
45. Scholz P (2005) Softwareentwicklung eingebetteter Systeme. Springer
46. Edwards L (2006) So, you wanna be an embedded engineer. Newnes
47. von Hagen W (2006) The definitive guide to GCC. Apress
48. Da Cruz F (1996) Using C-Kermit. Digital Press
49. Alonso G, Casati F, Kuno H, Machiraju V (2004) Web services. Springer
50. Heise Verlag (2003) c't special 02/03 Handhelds. Heise
51. ComputerEcology. http://computerecology.org
52. Embedded.com. http://www.embedded.com
53. Handhelds.org. http://www.handhelds.org
54. IBM developerWorks. http://www.ibm.com/developerworks
55. LinAccess. http://www.linaccess.org
56. Index of documentation for people interested in writing and/or understanding the linux kernel. http://www.dit.upm.es/jmseyas/linux/kernel/hackers-docs.html
57. Linux-to-go.org. http://www.linuxtogo.org
58. The linux documentation project. http://www.tldp.org
59. Open usability. http://openusability.org
60. TuxMobil. http://tuxmobil.org
61. Wikibooks. http://de.wikibooks.org
62. Wikipedia. http://de.wikipedia.org

Sachverzeichnis